ETL-0548

AD-A214 481

Vision-Based Navigation and Parallel Computing
First Annual Report

Larry S. Davis
Daniel DeMenthon
Thor Bestul
David Harwood

University of Maryland
Center for Automation Research
College Park, Maryland 20742-3411

August 1989

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

U.S. Army Corps of Engineers
Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060-5546

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S):	5. MONITORING ORGANIZATION REPORT NUMBER(S): ETL-0548
6a. NAME OF PERFORMING ORGANIZATION: University of Maryland; 6b. OFFICE SYMBOL (If applicable): N/A	7a. NAME OF MONITORING ORGANIZATION: U.S. Army Engineer Topographic Laboratories
6c. ADDRESS (City, State, and ZIP Code): Center for Automation Research, College Park, MD 20742-3411	7b. ADDRESS (City, State, and ZIP Code): Fort Belvoir, VA 22060-5546
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Defense Advanced Research Projects Agency; 8b. OFFICE SYMBOL (If applicable): ISTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DACA76-88-C-0008, ARPA Order No. 6350, Program Code No. 8E20
8c. ADDRESS (City, State, and ZIP Code): 1400 Wilson Blvd., Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 62301E; PROJECT NO.; TASK NO.; WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

VISION-BASED NAVIGATION AND PARALLEL COMPUTING -- First Annual Report

12. PERSONAL AUTHOR(S)
Larry S. Davis, Daniel DeMenthon, Thor Bestul, and David Harwood

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Annual	FROM 5/88 TO 5/89	August 1989	54

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	autonomous navigation, computer vision, parallel processing, search. (KP)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes research performed during the period May 1988-May 1989 under DARPA support. The report contains discussion of four main topics:

1. On-going research on visual navigation, focusing on a system named RAMBO, for the study of robots acting on moving bodies,
2. Development and implementation of parallel algorithms for image processing and computer vision on the Connection Machine and the Butterfly,
3. Development of parallel heuristic search algorithms on the Butterfly that have linear speedup properties over a wide range of problem sizes and machine sizes,
4. Development of Connection Machine algorithms for matrix operations that are key computational steps in many image processing and computer vision algorithms.

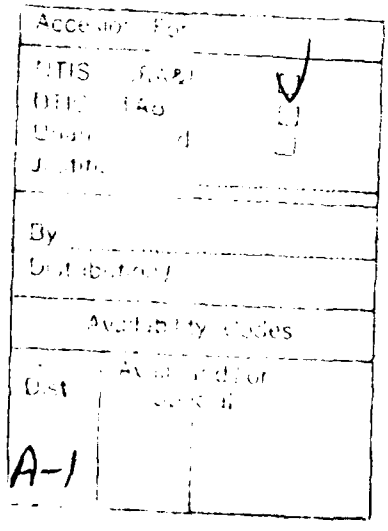
This research has resulted in twelve technical reports, and several publications in conferences and workshops.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Linda Graff	22b. TELEPHONE (Include Area Code): (202) 355-2818; 22c. OFFICE SYMBOL: CEETL-RI-T

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

PREFACE

This report describes work performed under Contract DACA76-88-C-0008 by the Center for Automation Research, University of Maryland, College Park, Maryland for the U.S. Army Engineer Topographic Laboratories (ETL), Fort Belvoir, Virginia, and the Defense Advanced Research Projects Agency (DARPA), Arlington, Virginia. The Contracting Officer's Representative at ETL is Ms. Linda Graff. The DARPA point of contact is Dr. William Isler.



Contents

1. Introduction 1

2. Navigation 2

2.1. RAMBO: Vision and Planning on the Connection Machine 2

2.1.1. Overview 2

2.1.2. Experimental Set-Up 4

2.1.3. Pose Estimation 7

2.1.4. Connection Machine Implementation 8

2.1.5. Task and Trajectory Planning 9

2.1.6. Perspectives 11

2.2. Zero-Bank Algorithm with Global Optimization 11

2.2.1. Background 11

2.2.2. The New Approach 12

2.2.3. The Algorithm 12

2.2.4. Experiments 14

2.2.5. Conclusions 14

3. Parallel Vision 16

3.1. Multiresolution Techniques 16

3.1.1. Pyramids and Hough Transform on the Connection Machine 16

3.1.2. Fast Addition on the Fat Pyramid 19

3.1.3. Replicated Image Processing 21

3.2. Quadtrees on the Connection Machine 23

3.3. Benchmarking Activities 25

3.3.1. Border Tracking with an Implementation on the Butterfly Parallel Computer 25

3.3.2. Parallel Matching of Attributed Relational Graphs 28

4. Parallel Iterative A* Search 31

4.1. Overview 31

4.2. The **A*** Algorithm 31

4.3. The **PIA*** Algorithm 32

4.3.1. Data Structures 32

4.3.2. Iteration Threshold, Mandatory Nodes and Speculative Nodes 32

4.3.3. The Node Expansion Procedure 33

4.3.4. Determining $\mathbf{t}^{(i)}$ 33

4.3.5. The Successor Distribution Algorithm 33

4.3.6. The Node Transfer Procedure 34

4.3.7. Termination 34

4.4. Analysis 34

5. Parallel Matrix Operators 35
5.1. Generalized Matrix Inversion on the Connection Machine 35
5.2. Grid Evaluation-Interpolation on the Connection Machine 35
5.2.1. Grid Evaluation-Interpolation using Tensor Products and General Inversion 35
5.2.2. Grid Evaluation-Interpolation using Multivariable Spline-Blending Approximation 36

6. Conclusions and Future Research 37

List of Figures

1 Vision-based control loop for a robot acting on a moving body. 42

2 Range Scanner System 43

3 A tagging joint takes a reaching trajectory during time T to reach a goal trajectory required to grip a handle on the target. 44

4 The cross-segment of the world road is assumed horizontal and perpendicular to the tangents at its end points. The tangents are assumed parallel. A condition satisfied by the matching points in the image which also involves the image tangents and the vertical direction is deduced. 45

5 Results of range/video fusion algorithm showing (a) the epipolar arcs drawn withing the ERIM image, (b) profiles showing the camera pixel ray and the elevation derived from the ERIM data along an epipolar arc, and (c) a perspective view of the resulting scene model. 46

6 Results on a slightly curved, upsloping road scene. (a–c) show the fusion algorithm compared with: (a) the flat-earth algorithm, (b) the modified zero-bank algorithm, and (c) the hill-and-dale algorithm. An overhead view of all four approaches is shown in (d). 47

7 Layout of the nodes of a 4-level pyramid mapped onto virtual processors linked by a Boolean 6-cube. 48

1. Introduction

This report describes research conducted during the period May 1988 - May 1989 under DARPA support. Our research during this past year has focused on four main areas:

1. Core research on autonomous visual navigation. Here, we are developing a system called RAMBO. RAMBO is an acronym for Robot Acting on Moving Bodies. RAMBO performs complex visual surveillance tasks on a moving object. In order to accomplish its tasks RAMBO must establish a model for the motion of its target, monitor and update that model over time, and plan and execute a trajectory that will allow it to illuminate specific points on the surface of the target with a laser. RAMBO is being implemented on the Computer Vision Laboratory's Connection Machine. Our progress on this project is described in Section 2.

2. Parallel vision algorithms. In order for RAMBO to successfully complete its tasks it must solve a variety of vision problems. If systems such as RAMBO are to operate in real time, then these vision problems must be solved using parallel algorithms. We are focusing on the effective utilization of the Connection Machine for solving vision problems at all levels of analysis—low level vision (preprocessing, local feature extraction), intermediate level processing (matching, stereo, motion) and high level processing (recognition and planning). In Section 3 we describe or research on parallel vision algorithms. This research includes general multiresolution image processing methods for the Connection Machine, as well as quadtree algorithms.

3. Search plays a central role in almost all Artificial Intelligence Systems, and especially at higher levels of analysis in vision systems. We have developed a new parallel search algorithm for MIMD machines, and have implemented this algorithm on the Laboratory's Butterfly. Experiments with the algorithm have resulted in linear speedups up to large numbers of processors (96) on realistic scheduling problems. The algorithm and some representative experimental results are presented in Section 4.

4. Parallel matrix operators. Many complex image processing operations, including restoration and reconstruction from projections, are based on common matrix operations (multiplication, inversion, etc.) We have developed several Connection Machine matrix manipulation algorithms. They are described in Section 5.

Our research during the first year of the current contract has resulted in twelve technical reports. The titles and abstracts of these reports are included at the end of this annual report. We also present, in Section 6, some brief remarks concerning our goals for the coming year.

2. Navigation

2.1. RAMBO: Vision and Planning on the Connection Machine

2.1.1. Overview

We now describe research being performed on a system named RAMBO (**R**obot **A**cting on **M**oving **BO**dies). RAMBO resides principally in the "mind" of a Connection Machine, and drives a monocular camera and laser pointer (attached to a robot arm) through space. A second robot carries a target (perhaps along a "virtual" surface to mimic the motion of a vehicle along the ground) attached to whose surface are sensors (light-sensitive diodes) with focusing optics. RAMBO's general task is to illuminate a set or sequence of these sensors for specific durations of time, possibly subject to overall temporal constraints.

We briefly describe the functions of the different modules of this system, from data collection to robot motion control.

1. The digitizer of the video camera mounted on the robot arm can obtain video frames when new visual information is needed.

2. A low level Connection Machine vision module extracts the locations of the projections of model features (i.e., polyhedral vertices) from the image. Once a model for the target's motion has been established, the predicted locations of visible target features are established using fast table lookup procedures implemented in the Connection Machine.

3. An intermediate level vision module establishes the instantaneous pose (location and orientation) of the target in the camera coordinate system.

4. The target motion predictor fits a target trajectory in location/orientation space to the most recent history of instantaneous pose estimates. The trajectories of so-called *goal points* around the target (called *goal trajectories*) are also determined. A goal point is a location fixed in the frame of reference of the target that one of the robot joints has to follow in order to accomplish one of the basic illumination subtasks of the total task. The determination of these trajectories should ideally take into account the subsequent visibility of a sufficient number of target features to verify or modify RAMBO's model of the target's motion, and safety criteria that would allow RAMBO to move away from the target to a safe viewing location in the event that the target's motion changes in an unanticipated way.

5. From the predicted goal point trajectories, the robot motion planner calculates the robot motions necessary for following the goal trajectories, and the resulting camera trajectories. If the subtasks were not ordered in the original task specification, then the motion planner orders them (using either optimal or heuristic methods). The camera trajectories are used for transforming subsequent target pose estimates from the camera coordinate system to an absolute coordinate system. In our current implementation, the Connection Machine is used to plan a smooth motion from one goal trajectory to a subsequent goal trajectory (from which the next subtask can be performed).

The RAMBO project thus provides us with a context for studying several basic classes of problems in vision and visual navigation. These problems include the development of parallel Connection Machine algorithms for efficient image processing and analysis, visual tracking, and visual planning.

In order for RAMBO to complete even its most basic navigation task, it must be capable of visually tracking its target through space. Feasible tracking algorithms depend on many factors including sensor field of view, processing time per frame, relative motion of the target and the sensor, accuracy of sensor control, etc. We can identify two basic approaches to visual tracking that are relevant to RAMBO:

1. Two-dimensional tracking algorithms, in which the target can be kept in the field of view by determining its image motion, and computing a suitable sensor motion that "nulls" or minimizes the image motion.

2. Three-dimensional tracking, in which a complete three-dimensional (3-D) model of the target's relative rigid body motion is established, and a sensor motion is determined that would cause certain components of the relative motion to be zeroed (for example, when RAMBO is firing its laser at the target, we might want all components of the relative rigid body motion between RAMBO and the target to be zero).

While approach (1) is simpler, it is not always applicable or sufficient. Its applicability depends on the factors listed above (i.e., sensor field of view, frame processing rate, etc.), although little research has been conducted that reveals the conditions under which two-dimensional (2-D) methods can be used for tracking. For example, given a frame processing rate, a model for the accuracy of image motion estimation and a model for the control accuracy of the sensor, one would like to know the minimum field of view that would guarantee (with some probability) that the target could be tracked.

However, even if it is possible to keep the target in the field of view using two-dimensional methods, it might not be sufficient for the overall planning process. For example, RAMBO should compute its trajectory through space based not only its particular sequence of illumination tasks, but also based on its ability to retreat if the target changes its motion in a way that might potentially cause it to collide with RAMBO. This type of analysis can be more easily accomplished using direct three-dimensional models.

One standard approach for three-dimensional tracking is to compute a sequence of instantaneous target pose estimates, and then to fit a motion model to this sequence of pose estimates. The pose estimation problem arises in other applications as well (e.g., object recognition), so is of general interest. In Section 2.3 we describe a Connection Machine pose estimation algorithm. This algorithm, based on methods previously developed in our laboratory, involves generating target pose hypotheses from matches of triples of image features to triples of target surface features, and clustering appropriate low-dimensional projections of the complete six parameter pose estimates. The algorithm is very fast, generating a pose estimate in the order of one second of Connection Machine time.

The overall computational architecture of RAMBO has not been finalized yet, but should be quite similar to the one we previously developed for road and road network navigation with separate computational modules for image processing, geometric reasoning, sensor control,

motion planning and plan supervision. RAMBO's task set, however, leads to a much richer set of problems in visual planning. In Section 2.5 we describe how RAMBO determines an appropriate subtask ordering if the initial task definition does not specify a fixed subtask ordering, and also explain how the Connection Machine can be used to establish smooth motions between consecutive goal trajectories.

2.1.2. Experimental Set-Up

We mounted a CCD camera and a low-power laser pointer on the tool plate of the *American Cimflex* robot arm. Images from the camera are digitized and can be sent to the Connection Machine for processing. We purchased a smaller robot arm (*Mitsubishi RM-501*), which can translate and rotate an object (the target) through space. Several light-sensitive diodes with focusing optics are mounted on the surface of the object. RAMBO's goal is to hit a sequence of diodes on the moving object with its laser beam for given durations, possibly subject to overall time constraints. Electronics inside the object monitor the status of the diodes and transmit this status to a computer through a serial line.

Simultaneously, we are developing a full computer simulation in which the camera inputs are replaced by synthetic images. Figure 1 is a schematic representation of the equipment. This figure also shows the vision-based control loop—using actual and simulated images.

We also built a Fast Range Scanner, which has various applications for navigation and calibration within the RAMBO project. The range scanner design is based on the following known principle:

- A sheet of light is scanned across the scene in small increments, so that at the end of the scan all the points of interest in the scene have been illuminated.
- For each scanning position of the sheet of light, a camera collects an image of the scene containing the stripe of light created by the sheet.
- In each camera image obtained from previous step, and for each image pixel which belongs to the centerline of a stripe, the distance of the corresponding point which is illuminated by the sheet can be obtained, because such a point is at the intersection of the sheet of light and the line of sight given by the pixel. This step is repeated for all pixels on the centerline of a stripe.

In 1987, we designed and built a range scanner based on these principles. The two main problems with that scanner were the very long time of range image acquisition (around 5 minutes) and the poor precision of the range image obtained, due to the thickness of the light stripes created by a remote light source and a fiberoptics light guide.

The goal of this project was to build a range scanner with better precision and faster acquisition speed. The precision is improved by the use of a laser source and components mounted on an optical table. The speed-up is obtained by building hardware specialized for the detection of the stripe pixels.

2.1.2.1. Optical Components

The range scanner system contains a laser source with optics, a mirror rotated by a stepping motor and a camera. It is illustrated in Figure 2.

The laser source is a 1 mW helium-neon laser. It is equipped with optics transforming the laser beam into a thin sheet of light. The sheet of light spreads around 7.5 degrees on each side of the original beam axis.

The mirror is used to scan the sheet of light across a scene. The mirror is mounted on the axis of a stepping motor block. The stepping motor block includes a harmonic gear with virtually zero backlash. The step angle is 0.0144 degrees.

The camera does not use a CCD sensor. Instead, a light-sensitive RAM (Random Access Memory) is mounted on the image plane, and provides a digitized 256 × 128 one bit deep image directly. This "Optic RAM" is a normal 64 kbit dynamic RAM (D-RAM) covered by a transparent window, and light can hit the memory cells through this window and discharge these cells. In a D-RAM, all of the cells keep losing their charges, but in a cell which receives light, the discharge becomes more active. The optic RAM uses this feature. All cells are charged to 5V, then are read after an exposure time called *soak time*. The cells exposed to sufficient illumination have their voltage drop below a given threshold, and these locations correspond to an image pixel with the value 0, while the cells which kept their charges above the threshold are given the value 1. Then all the cells are charged again for a new soak time and a new image.

Compared to a CCD sensor, an optic RAM is both an image sensor and an image memory, resulting in a simpler circuit. A CCD camera would require a digitizing circuit and a frame buffer, and would store several bits per pixel, more than is needed to simply detect positions of stripes of light. The drawback of the only Optic RAM sensor available on the market is that it was not optimally designed for optical applications. It is very small (4.42 mm × 0.88 mm) compared to CCD chips, so that in order to get an acceptable field of view a lens with short focal length must be used. Such lenses have a tendency to absorb a lot of light and distort the image. Also, the width of the field is five times its height, whereas most CCDs give a field 1.5 wider than its height. We are investigating new light sensing components such as CIDs (Charge Injected Devices) which have similar advantages to the Optic RAM but larger dimensions.

2.1.2.2. Control Hardware

This hardware system consists of about 50 logic ICs, with a NuBus parallel interface to the Macintosh II computer. It includes the stepping motor controller, the Optic RAM controller, the stripe detector, the stripe buffer and a conversion table RAM. By using this circuit, we can control all of this system and get the desired range.

- **Optic RAM Controller**

The Optic RAM controller has a refresh counter, a row address counter and a column address counter. When the optic RAM is not exposed, we always refresh it. We use 2-phase, 1μs

clocks for system clocks and use a half-period for refresh, and another half-period for the read/write operation. We need 256μs to refresh the whole row of image cells. The pixel which is addressed by the row and column address counters can be read or written in this period. The hardware can access each pixel at a rate of 1 MHz.

The camera and laser source are mounted in such a way that stripes of light are almost parallel to rows of the RAM light-sensitive array. Each column is scanned until a pixel 0 is found, indicating that we reached the leading edge of a stripe. Then we keep scanning until we find a pixel 1 indicating that we reached the other edge of the stripe. We deduce the address of the centerline of the stripe half way between these two pixels. The hardware circuit repeats this process for each column of the array.

Now, for the first stripe in the scanning process we start the search from the edge of the array. But if we assume we swing the laser from left to right, the next stripe generally appears on the right of the previous one. We use this property to start looking for the next stripe from the leftmost pixels of the previous stripe. The whole process is implemented in hardware.

- **Stripe Buffer**

We use a 1 kbyte FIFO memory as a stripe buffer. The data coming at high speed from the stripe detector are fed to the FIFO and can be read more slowly.

- **Range Conversion Table**

A conversion table gives the range when the stripe address and the mirror angle are given. This table is calculated in the host computer prior to the range image capture from geometric parameters of the relative mirror-camera configuration and transferred to two 32kbyte RAMs of the Range Detector board.

2.1.2.3. Operations

For a 256×128 range image, a sequence of around 256 images must be processed, each with a stripe at a slightly shifted position, created by a new angle of the mirror shifted by 0.0144 degrees with respect to the previous step. One step of this process consists of moving the mirror, soaking the Optic RAM, detecting the positions of the centerline pixels of the stripe, reading the corresponding ranges in the conversion table, and transferring the range data.

At first we move the mirror from the home position to the first angle, and start soaking the Optic RAM, i.e. we stop refreshing the optic RAM. Then we start detecting a stripe, while moving the mirror to the next angle at the same time. The time for the stripe detection depends on the shape of the stripe, and is of the order of 10 ms. This time becomes longer when we can't detect a stripe (it can be hidden behind an object). The time for moving the mirror and letting it settle does not exceed 10 ms.

Then we can start to soak the Optic RAM for the next stripe. Meanwhile we start transferring the range data to the computer. The transfer time is very short compared to the soak time, because we use a NuBus parallel interface, which is about 50 times faster than

the RS232 serial interface of the Macintosh II computer. With a lens of 10 mm focal length, we need 10 to 20 ms for soaking, and 4 to 10 seconds to get the whole range image. But when we use a lens of 4.8 mm focal length, to get a wide field (50 degrees × 10 degrees), we need about 70 ms for soaking because this lens absorbs, and about 20 seconds are required to get the whole range image.

Prior to capturing a range image, we can run a self-diagnostic routine, which checks the Optic RAM, the Stripe Detector and the Range Conversion Table.

2.1.3. Pose Estimation

We are presently exploring Kalman filtering techniques for predicting future poses of a polyhedral target from past estimates. Using these techniques in a "feed-forward" mode of processing, we can quickly compute a new pose estimate based on predictions of the image projections of specific target surface features. However, in order to "bootstrap" this procedure, or to recover from gross errors due to either changes in the trajectory of the target or mistakes in image analysis, we have developed a Connection Machine pose estimation algorithm that is not based on any prior knowledge of target pose or motion. This algorithm is based on work originally performed by Linnainmaa, Harwood and Davis in our laboratory. In that paper, we showed that if a triple of image features (i.e., perspective projections of polyhedra corners) could be matched to a triple of target surface features, then a simple quartic equation can be solved to determine a small number of six degree of freedom pose estimates (in fact, the equations almost always have only two solutions). Since it is difficult to determine which image features match to which target features in the absence of prior knowledge, the basic hypothesis generation procedure is embedded in a clustering algorithm that matches many combinations of three image features against combinations of three target features. Various heuristics can be employed to reduce the combinatorics of this matching process. The key to the success of the clustering process is the choice of an appropriate projection of the six-dimensional pose space in which to perform the initial clustering. The projection used was a two-dimensional projection corresponding to the visual direction to the target center under any hypothetical pose estimate. Within each bin of this two-dimensional clustering space, pose estimates were grouped by visual size of the target.

A straightforward implementation of this algorithm on the Connection Machine would result in a very slow pose estimation process because of the intensive floating point arithmetic operations associated with solving the quartic equations determined by each image triple/target triple combination.

Our Connection Machine pose estimation algorithm combines three ideas:

- Pose estimation by matching triples of image features to triples of target features.
- Standard camera rotations.
- Paraperspective approximation to perspective projection.

This combination allows the extensive use of lookup tables in lieu of costly floating point arithmetic operations. Feature points detected in the image are grouped into triples (called

image triangles). Each image triangle is defined by a distinguished vertex (called the *reference vertex*), the lengths of the two adjacent sides, and the angle between them (the *reference angle*). Image triangles can be computed for all triples of detected image features (three triangles per triple, since any point can be chosen as the reference vertex) or various heuristics can be employed to reduce the number of image triangles constructed (for example, our current implementation includes a simple test for vertex connectivity).

The determination of the target's pose is somewhat simplified if each image triangle is first transformed so that its reference vertex is at the image center and one of its edges is coincident with the image x-axis. This is equivalent to a rotation of the image plane (with the rotation parameters expressed as functions of the reference vertex's initial image position) to bring the reference vertex to the image center, followed by a camera roll to bring one edge into coincidence with the x-axis. Kanatani developed simple formulas for these rotations. Combining these techniques, we compute the location of the centroid of the target, and then compute its projection onto the image plane for each pose hypothesis. We then apply the inverse transformations of the roll and standard rotation (that brought the reference vertex to the image center to the target centroid projection) to compute the direction of sight of the target under this particular pose hypothesis. A two-dimensional array of possible centroid projections is maintained, and at each location in the array we count the number of image triangle/target triangle pairs that yielded pose estimates resulting in that projected target centroid location, and we maintain a list of those pairs along with the distances computed to the target centroid. This information is used by a subsequent clustering algorithm to identify large subsets of image triangle/target triangle pairs yielding sufficiently similar pose estimates.

2.1.4. Connection Machine Implementation

The Connection Machine is used in three ways to implement the pose estimation algorithm:

1. as a lookup table engine;
2. as a combinatorial machine, considering all combinations of image triangles and target triangles;
3. as an image processor for calculating convolutions and finding peaks (clustering) in the Hough transform space.

2.1.4.1. Lookup Table Engine

There are several different table lookup operations performed in the course of pose estimation. First, the parameters of the standard rotation are stored in a two-dimensional lookup table indexed by image position. The parameters of the inverse rotations are also stored in this two-dimensional lookup table.

A second set of lookup tables is maintained, one for each possible target triangle. These are also two-dimensional lookup tables, indexed by α, the reference angle of an image triangle and K, obtained by dividing the ratio of the two image triangle sides adjacent to α and the

ratio of the two corresponding target triangle sides. Each such table contains the three-dimensional location of the target centroid as output (there is no need to explicitly store or compute the intermediate variables corresponding to the orientation and location of the target triangle in the image coordinate system).

All of these tables can be computed beforehand and loaded into the Connection Machine.

2.1.4.2. Combinatorial Machine

Here we describe how the image triangle/target triangle pairs are distributed in the Connection Machine. We regard the Connection Machine as a two-dimensional array. Each target triangle is assigned to one row of this array. The information initially associated with each image triangle includes the image plane coordinates of its vertices and the parameters of both the standard rotation and its inverse. The target triangle data is then scanned across the rows and the image triangle data is scanned up the columns to create the combinatorial pairing of image triangles/target triangles.

2.1.4.3. Image Processor

The analysis of the pose estimate voting patterns of the image/target triangle pairs involves operations common to basic image processing. The two-dimensional clustering array of projected target centroids is represented in the Connection Machine by assigning one processor per location of this two-dimensional array. After all votes are cast by the image/target triangle pairs, the counts in this array are locally smoothed, and the smoothed array is then thresholded. The above threshold processors are then numbered according to their vote strength, and the image/target triangle pairs that contributed to the above threshold counts are selected for further processing.

A second clustering step is then applied, in parallel, to the triangle pairs corresponding to each above threshold centroid projection. Each centroid projection is assigned to a row in a two-dimensional matrix, and the triangle pairs that contributed to that centroid projection are then loaded into the columns of that row and bucket sorted by Z coordinate of the centroid. Each row is smoothed independently, and the highest cluster is finally selected as the correct cluster. The triangle pairs that contributed to that cluster are then selected, and a final least squares estimate of the target's pose is computed based on the actual correspondence of image features to target features.

2.1.5. Task and Trajectory Planning

In order to perform task and trajectory planning, RAMBO currently makes the simplifying assumption that a complex goal can be decomposed into a sequence of simple subgoals, and that each subgoal can be performed with one joint of the robot in a fixed position with respect to the target. This joint has to "tag along" with the target and so we call it the *tagging joint* of the robot. The fixed position with respect to the target that the tagging point must follow to complete a subgoal is called the *goal point*. All goal points required for each complex action on a target can be predetermined and stored in a database of actions specific to each target. Each goal point is defined by its pose in the target coordinate system.

As RAMBO proceeds from one subgoal to another, it will generally have to change the trajectory along which it moves, so that at some time t_o we would want RAMBO to launch from its current goal trajectory and to land at some subsequent time, $t_o + T$, on a new goal trajectory. The duration T is referred to as the *reaching duration*. Once t_o and T are chosen, then a *reaching trajectory* that takes RAMBO from its original goal trajectory to the new goal trajectory can be determined by using, for example, a parametric cubic spline that ensures continuity and smoothness at both takeoff from the original trajectory and landing on the new goal trajectory. See Figure 3.

A subproblem, then, in controlling RAMBO's motion from one trajectory to another is the choice of T, the transit time. It should be chosen so that the resulting linear and angular velocities and accelerations are within the limits of RAMBO's motions. Additionally, the resulting reaching trajectory should be safe in the sense that it should not cross the path of the target and should not require RAMBO to assume impossible configurations. The Connection Machine can be used to examine a range of reaching durations in parallel, finally choosing the smallest reaching duration resulting in a realizable reaching trajectory.

We set up a two-dimensional array of processing cells with time as the vertical dimension, and values of T as the horizontal dimension. Each column of the array contains a copy of the predicted goal trajectory, with the first row containing the position of the goal at the present time in location/direction space, the next row containing the position at a time increment beyond that, and so on. Every column also contains a copy of the trajectory of the tagging joint from the original trajectory, sampled with the same time increments as the goal trajectory. The difference between columns is that they use different durations, T, of the reaching trajectory, increasing from one column to the next.

Each cell in the array computes a point of the reaching trajectory for the time t_o corresponding to its row and for duration T corresponding to its column. It then computes estimates of appropriate derivatives of its reaching trajectory by communicating with its neighbors in the column. The maxima of the derivatives are computed for each column and the column that has the smallest T for which the maximum derivatives are within bounds is chosen to determine the reaching trajectory. The near term future motion of the robot should be controlled based on this selected trajectory.

A rule-based system is used for completing sets of tasks. It is based on a heuristic strategy for dealing with illegal trajectories (ones involving collisions or impossible robot positions or motion derivatives) and incorporates either a greedy strategy for choosing tasks or a fixed task ordering. Both versions are described below. Essentially, the following ordered set of rules is iteratively applied until the set of tasks is completed. The rules were chosen to be simple, but fairly robust.

Rule 1: If currently on the goal trajectory of an uncompleted task, remain on it for the specified task duration, then restart rule set.

Rule 2 (Greedy): Find reaching trajectories to all remaining goal trajectories. Choose quickest legal reaching trajectory to a goal trajectory and follow it until reaching the corresponding goal trajectory, then restart rule set.

Rule 2 (Fixed order): Otherwise, find reaching trajectory to the goal trajectory of the

next task in the desired sequence. Follow this reaching trajectory until reaching the corresponding goal trajectory, then restart the rule set.

Rule 3: If there is no legal reaching trajectory to a goal trajectory, find a reaching trajectory to the "approach" trajectory (this is a pre-defined trajectory relative to the target motion from which some goal trajectory is likely to be reachable in the future), begin to follow this reaching trajectory and restart the rule set.

Rule 4: If the reaching trajectory to the approach trajectory is not legal, maintain present position relative to the target and restart the rule set.

If a revision of the target motion model parameters occurs during the application of any of the rules, the rule set is restarted. This is because a revision of the target motion will sometimes cause a reaching trajectory or section of goal trajectory previously thought to be legal to be disallowed. Most of the time, however, since the revisions of the motion model parameters will not be large, there will be no drastic change in the robot motion due to them.

2.1.6. Perspectives

This is an on-going project. We have described progress to date on constructing a set of Connection Machine vision and planning algorithms that should allow RAMBO to plan, monitor and execute a complex navigation task. These algorithms are currently being integrated so that they may be tested on some simple initial navigation tasks. Additionally, we are developing fast Connection Machine two-dimensional target tracking algorithms, which could be interleaved with the more computationally demanding three-dimensional tracking algorithms, and studying the applicability of logic programming and probabilistic reasoning methods for specifying, synthesizing, monitoring and controlling RAMBO's actions. Details will be given in future reports.

2.2. Zero-Bank Algorithm with Global Optimization

We developed a new "zero-bank" method for the reconstruction of a road in three-dimensional space from a single image. The world road is modelled in a similar way as in our previous "zero-bank" method, i.e. as a space ribbon generated by a centerline spine and horizontal *cross-segments* of constant length (the *road width*) cutting the spine at their midpoint at a normal to the spine. Because reconstructions of cross-segments are now independent of each other, a global optimization of the road reconstruction can be used.

2.2.1. Background

Our previous zero-bank method was recursive, requiring the knowledge of a previous cross-segment of the road to compute a new cross-segment. The problem was that at each recursion step, up to three possible cross-segments were found by solving a cubic equation; each of these three segments could be used as starting elements for up to three new cross-segments

at the next recursion step, leading to a tree of possible roads growing exponentially. To avoid exploring all these alternatives, we chose in our previous method to keep at each step only the most plausible cross-segment, namely the cross-segment giving the smallest changes of road slope and turn. These are local criteria, however, whereas the goal is to obtain the best *global* road reconstruction. The images of the road edges could be locally of poor quality, leading to the wrong branching choice at this place of the tree of possible reconstructions, and the rest of the construction downstream could suffer or be brought to a dead end.

2.2.2. The New Approach

In the new method, tangents to the road edges at the end points of cross-segments are assumed to be approximately parallel. Thanks to this reasonable addition to the road model, a recursive reconstruction is not required. Cross-segments can be found from any element of road image, independently of previous pieces of reconstruction. Several possible local solutions of cross-segments are still found, but in this case we do not need to make choices until all the available evidence from the image has been used. Then we can choose the best global reconstructed three-dimensional road passing through the alternative reconstructed cross-segments. This global optimization is not computationally expensive because we can apply a dynamic programming technique minimizing local slopes and turns.

2.2.3. The Algorithm

A summary of the algorithm is now given; Figure 4 illustrates the road geometry reconstruction.

1. In a preliminary step, not detailed here, some appropriate image processing techniques have isolated the two curves of the edges in the image, and a polygonal approximation has been found for each edge curve.

2. Picking image points anywhere on one image edge curve, we are able to find the points which are candidates for being *matching points* on the other image edge curve. (Two image points are called matching points if they are images of the end points of cross-segments.) We found an expression that two image points located on the facing image edge curves and the tangents to the edge images must satisfy to be matching points. If a_1 and a_2 are matching points and $\vec{a_1}\prime$ and $\vec{a_2}\prime$ are the tangent directions to the image edges in these points, the following relation holds:

$$[\vec{V} \times (\vec{a_1} \times \vec{a_2})] \cdot [(\vec{a_1} \times \vec{a_1}\prime) \times (\vec{a_2} \times \vec{a_2}\prime)] = 0 \qquad (1)$$

   For edge curves approximated by polygonal lines, the matching point a_2 can be on a line segment, and its position between the end points of the line segment can be expressed by a number between 0 and 1, whereas its tangent vector $\vec{a_2}\prime$ is constant ; or the matching point a_2 can be at an end point of a line segment, with a constant position but with a tangent angle which can be expressed by a number between 0 and 1 within the range of angles of the two adjacent line segments. For one point picked on one image edge, we check for each of the line segments of the other image edge

if a matching point belongs to that line segment, i.e., if our expression gives a linear coordinate between 0 and 1 for this line segment. Then we look for matching points at the nodes of the polygonal line by checking if the expression gives a number between 0 and 1 for the tangent angle. We repeat these searches for several points a_1 picked on one image edge.

3. For each point picked on one edge image, the previous step can give several matching points on the other edge image. One of the reasons is that the images of the edges can be very rough and wiggly. Another reason is that the condition used is only a necessary condition for two points to be matching points in the image of the road we are seeing. This condition is local and it is up to us to pick up among the found matching points the pairs which are the most globally consistent, and discard the other pairs. The criteria of optimization are three-dimensional criteria; thus at this step of the algorithm, from the pairs of matching points, the corresponding three-dimensional cross-segments must be found. This correspondence is unique if the cross-segments are assumed horizontal and of known constant length. The constant length is the width of the road, and cannot be defined by this method. The assumed road width is a scaling factor in the reconstruction, whereas the optimization is based on angular considerations, which are independent of scaling. For driving a vehicle the road width must eventually be obtained from other methods, such as stored data about the road, the "Flat Earth" method, or close-range methods such as stereoscopy or time-of-flight ranging.

4. The group of matching point pairs corresponding to a single point chosen on one edge is the image of a group of world cross-segments obtained at the previous step, and the world road can go through *at most one* of these cross-segments. If a sequence of points along one road edge is taken, a sequence of groups of cross-segments is obtained, and the world road must go through at most one of the cross-segments of each group, in the same order as the sequence of points chosen on the first road image edge. Each cross-segment can be represented by a node of a graph. A path must be found in the graph, which visits each group in the proper sequence and goes through at most one node of each group, and which maximizes an evaluation function which characterizes a "good road". The total evaluation function is the sum of the functions of each of the arcs of the graph. The evaluation function for an arc is the sum of weighted criteria, which grade the choices of individual cross-segments and the neighborhood of consecutive cross-segments, based on angular considerations. It would also seem useful to introduce constraints such as a requirement for small differences of slope between successive patches, but this type of relation involves three successive cross-segments and complicates the interaction graph. The previous unary and binary criteria actually appear to be sufficient for discarding unwanted nodes. Dynamic programming is appropriate for this type of path optimization, and, compared to a brute force search for the best path, greatly reduces the complexity of the search.

2.2.4. Experiments

Two types of experiments were performed: (1) reconstructions of roads from synthetic road images and comparison with the road models used to create the synthetic images; and (2) reconstruction of roads at Martin Marietta, Denver, from video data obtained from the ALV, with comparisons of the results with the reconstructions obtained by data fusion between video data and ERIM scanner range data.

In the synthetic data experiments, a criterion of *navigability* of the reconstructed road was defined, as the percentage of actual visible road that a vehicle half the width of the road can follow without driving its wheels on the edge, if the vehicle were following the reconstructed road given by the algorithm. For all configurations of slopes and image noise tested, the navigability of the reconstructed road is significantly better with the new zero-bank method than with the previous method or the Flat-Earth approximation.

In the experiment with actual road images at Martin Marietta, Denver, the "ground truth" was given by a method combining range data and video data, developed by Morgenthaler and Hennessy. The road edges are detected in the video image. Considering the line of sight of a road edge point, the world point of the edge is the point where this line of sight intersects the ground. The line of sight has a range image (*epipolar curve*) which can be calculated and superposed on the range image of the ground. The intersection of the line of sight with the ground corresponds to a point on the epipolar curve point with the same range as the ground point of the ground range image. From the ranges of several road edge points a three-dimensional profile of the road edges is obtained. This is illustrated in Figure 5.

Reconstructions were produced for around 50 road configurations including combinations of turns and slope changes. Both methods proved quite robust, giving plausible and consistent road reconstructions for all these tests; however, the ERIM laser ranger has a limited range of action. Only the first 15 meters of the road could be reconstructed by the fusion method. The reconstruction by the zero-bank algorithm extended at least twice as far in most road configurations. In the short stretch where both reconstructions were available, the agreement was considered good in top view (Figure 6). Differences of elevations appeared in side view, although the difference would probably not have resulted in different steering of the vehicle. The zero-bank algorithm used a flat earth approximation in the first segments of the road to remove the scale-range ambiguity inherent to this algorithm. The flat-earth plane is calculated as an extension of the plane of the points of contact of the vehicle wheels with the ground. If the vehicle is on a local bump, the actual road will be lower than the result of this approximation. The laser ranger will of course detect this lower profile, and the fusion algorithm should produce a correct road profile.

2.2.5. Conclusions

Experiments with synthetic data show that the new zero-bank method gives useful information about road profiles even far away from the vehicle. Experiments with real data and comparisons with road reconstructions obtained by fusion between video data and range data show a good agreement in the short range in which range data are available, provided the scaling factor not defined by the zero-bank reconstruction is well chosen. Until range

scanners covering a field as large as the field of view of video cameras are developed, the two methods can profitably be used together. The fusion algorithm can give the ground truth on the first 15 meters in front of the vehicle. The zero-bank algorithm can use this ground truth to remove its scale-range ambiguity, and extend the road reconstruction to most of the camera field of view. References and details are provided in [1].

3. Parallel Vision

3.1. Multiresolution Techniques

3.1.1. Pyramids and Hough Transform on the Connection Machine

3.1.1.1. Background

Pyramid architectures and algorithms have been demonstrated to be useful in fast computation of global properties by performing only local operations. Embedded in the rapidly-decreasing sizes of the processor arrays is a multiresolution data structure which enables parallel multiresolution image analyses. Since processors at different levels in a pyramid are connected in a tree structure, the height of which is only logarithmic in the image size, global information can be extracted in logarithmic time. Log(image diameter) time performance can be achieved for image filtering and segmentation, among other tasks. We developed a pyramid programming environment on the Connection Machine. Pyramid algorithms can be implemented in this programming environment to gain more knowledge about the performance of parallel pyramid algorithms. A pyramid Hough transform program based on a merge and select strategy among line or edge segments is presented to illustrate the capability of our system.

In the Connection Machine, groups of 16 processors are placed on single chips and connected by a 4 by 4 grid. Global communication is done via a communication network called the *router*. The router is a Boolean 12-cube with every router node connected to 16 processors. In addition, there is a communication mesh called the *NEWS* network which connects the processors in a two-dimensional mesh. Local grid communication between processors is expected to be faster via the NEWS network than via the router.

For applications requiring more processors than there are in the machine, the Connection Machine provides the mechanism of *virtual processors*. If the machine has 2^K physical processors while 2^N, $K < N$, processors are needed for the application, a mechanism is created so that every physical processor simulates 2^{N-K} virtual processors, each having an unique virtual cube address N bits long. In this way, a maximum number of physical processors remain active during program execution.

3.1.1.2. Design Considerations

Our system was designed so that pyramid architectures with rectangular support from children and resolution reduction factors of 2 could be implemented easily. Children linked to a father are not limited to be only those in a 2 by 2 block but can be in any rectangular block having even sides and centered at the 2 by 2 block. Such a construct is particularly helpful in designing pyramid structures with elongated support from children as well as for overlapped pyramids. Our system also provides a programming environment compatible with other existing vision software on the Connection Machine. Users need only define the data structure of a pyramid node and the local operations desired, both inter- and intra-level, to develop specific systems. Configuring the Connection Machine into a pyramid, setting up

the necessary linkages between processors, local memory allocation, and global bottom-up and top-down activation are all done automatically by our system.

3.1.1.3. Embedding Pyramids into the Connection Machine

Several methods have been proposed to map the pyramid architecture onto a hypercube architecture. Our mapping is based on a scheme called *Shuffled 2D Gray Code*. It uses *Reflexive Gray Coding*. Reflexive Gray Coding has been widely used in mapping hypercubes onto processor arrays as well as pyramid machines. It has the attractiveness that two successive codes differ by only one bit.

For the computation of the Shuffled 2D Gray Code, we first compute the reflexive Gray Code $G(i)$ and $G(j)$ of two integers i and j. We then *interleave* the bits of $G(i)$ and $G(j)$ to derive $SG(i,j)$, the Shuffled 2D Gray Code.

We identify every Connection Machine processor by its virtual hypercube address. Let *node* (i,j) represent the pyramid node on level l (the bottom level is level 0) with grid coordinates (i,j), $0 <= i,j < 2^{L-l}$, where L is the height of the pyramid. This node is mapped to the virtual processor with cube address $SG(i,j) * 4^l$. Every pyramid level therefore maps one to one onto the whole machine. The number of levels each virtual processor is involved in for this simulation is one plus half the number of trailing zeroes of its cube address. Physical processor 0 will simulate the greatest number of pyramid nodes, 13 in a 64K machine on a 512 by 512 image. See Figure 7.

There are several advantages to using this mapping scheme. First, every pyramid level in this mapping forms a 2D Gray Code mesh. Thus the whole pyramid architecture is homogeneous. Inter- and intra-level communication involves the same operation for every pyramid node and is independent of its location.

Secondly, because each row and each column in a 2D Gray Code mesh constitutes a Gray Code sequence, each of the cube addresses of the four mesh neighbors of any pyramid node differs from that of the given node by one bit only. This means that every 4-neighbor is only one communication link away while every 8-neighbor is only two communication links away. The communication cost between any neighboring pair of nodes on any level is greatly reduced.

Thirdly, determining parent/child relations is very easy. For processor $SG(i,j) * 4^l$, which is node (i,j) on level l, the cube address of its direct parent is $SG(i/2, j/2) * 4^{l+1}$. Computationally, this is equivalent to setting bits $2l$ and $2l+1$ of its own cube address to zero (the least significant bit is bit 0). The addresses of the four direct children, if they exist, can be determined similarly by setting bits $2l-2$ and $2l-1$ of its own address to 00, 01, 10, and 11, respectively. The direct parent of any pyramid node is at most two communication links away.

Our mapping scheme maps each pyramid level onto the whole Connection Machine. This is adequate for most pyramid algorithms where only one level of the pyramid is active at any time. For algorithms requiring that more than one level be active at one time, it is possible to map the whole pyramid onto the Connection Machine so that every neighbor is at most two communication links away, provided that the number of processors in the hypercube is at least twice the number of processors in the pyramid.

3.1.1.4. Pyramid Hough Transform

We show that a pyramid machine can be used to perform a variation of the Hough transform by implementing the following algorithm.

Every line in the X-Y plane can be represented by a pair of coordinates, (ρ, θ), where ρ is the distance from the origin to the line and θ is the directed angle from the positive X axis to the normal of the line. Every point (x, y) on this line has coordinates of the form $(\rho\cos\theta - t\sin\theta,\ \rho\sin\theta + t\cos\theta)$. The line segment from $(\rho\cos\theta - t_1\sin\theta,\ \rho\sin\theta + t_1\cos\theta)$ to $(\rho\cos\theta - t_2\sin\theta,\ \rho\sin\theta + t_2\cos\theta)$ can be represented as (ρ, θ, t_1, t_2).

Because of the exponentially tapering structure of the pyramid, information gathered from children needs to be condensed so as to be stored locally and passed up for further processing. We assume every pyramid node to have a bounded memory capacity able to store at most K 4-tuples (ρ, θ, t_1, t_2) representing line segments. Every node on level l collects from its four direct sons a maximum of 4K such tuples. It then tries to merge the line segments represented by these tuples into longer segments using a *distance* measure between segments. The K *best* segments surviving the merge are stored at that level-l node. This process is repeated at levels 2, 3, ... until the apex of the pyramid is reached. At this stage the apex has stored the K most prominent line segments in the image.

We measured the actual spatial distances between pairs of line segments in order to merge segments which are collinear and close to one another. We defined the distance between two line segments as the maximum of the distances from one of the endpoints of one segment to the line on which the other segment lies. This distance is computed for all pairs of segments collected at current node. The pair with the minimal distance is merged into one segment if the distance is below a certain threshold. The process is repeated for the remaining 4K-1 segments until either the threshold is exceeded or the number of segments remaining equals K.

To merge two segments into one, we define the new segment as lying on the line determined by the midpoints of the two pairs of endpoints of the two segments. The endpoints of the new segment are the two outermost of the projections of the original four endpoints onto this line.

Finally, to decide which K segments to keep, a node computes for every merged segment the sum of the line (or edge) strengths of all pixels contributing to that segment. Shorter and weaker lines or edges will have lower strengths and will be weeded out gradually as the process continues up the pyramid. This leaves the more salient ones at higher levels in the pyramid, as desired in a Hough transform.

In conclusion, our results show that the performance of the pyramid Hough transform is not much worse than that of the ordinary Hough transform except when the globally prominent lines or edges consist of collections of collinear short segments which are not locally salient. Further research is currently being conducted using our pyramid programming environment on the Connection Machine to explore further capabilities of both the Connection Machine and the pyramid architecture. Details and references are given in [2].

3.1.2. Fast Addition on the Fat Pyramid

We developed an algorithm for fast addition on a parallel fat pyramid, i.e. a fat pyramid in which each pyramid node has computational capabilities. A pyramid representation of a $2^n * 2^n$ image is a stack of successively smaller arrays of nodes: level l of the pyramid will have a size of $2^l * 2^l$, with the root of the pyramid being level 0. In the *fat pyramid*, the size of a node depends on the level of the pyramid in which it appears. A node at level l of the fat pyramid will have four times as much storage space and processing power as a node at level $l + 1$. In this paper we assume a fat pyramid in which larger processors are implemented through the use of aggregations of smaller processors.

Our addition algorithm is based on a *carry-lookahead* technique that has been proposed in the literature. It is based on the fact that *a carry does not depend explicitly on the preceding one, but can be expressed as a function of the relevant augend and addend bits and some lower-order carry.* The technique defines two auxiliary functions: $G_i = a_i b_i$ (the carry-generate function) and $P_i = a_i + b_i$ (the carry-propagate function); G_i gets the value of 1 when a carry is generated at the i^{th} stage, while P_i gets the value of 1 when the i^{th} stage propagates the incoming carry c_i to the next stage $i + 1$. The carry c_{i+1} can then be expressed as: $c_{i+1} = a_i b_i + (a_i + b_i)c_i = G_i + P_i c_i$ which implies $c_{i+1} = G_i + \sum_{j=0}^{i-1}(\prod_{k=j+1}^{i} P_k)G_j + \prod_{k=0}^{i} P_k c_0$.

If a single processor is allocated to a single node at level n, then 4^{n-k} processors will be allocated to a single node at level k. This implies that the operands for the addition operation at level k should be distributed throughout those 4^{n-k} processors. For reasons of uniformity, the same number of bits from each operand will be allocated to each of the above processors. The addition of two N-bit operands returns an $N + 1$ bit result; however, $N + 1$ may not be perfectly divided by 4^{n-k}. This implies that sign-extension of each operand should be performed to transform it into a number having the same value as before, but with a number of bits equal to a multiple of 4^{n-k}. We will be using the *two's complement representation* of binary numbers, so if $a_{N-1}a_{N-2}\ldots a_1a_0$ and $b_{N-1}b_{N-2}\ldots b_1b_0$ are the binary representations of the operands A and B respectively, we will allocate a total of $4^{n-k}\lceil (N+1)/4^{n-k}\rceil$ bits to each operand and the result, where $\lceil (N+1)/4^{n-k}\rceil$ of those bits will be stored in each one of the 4^{n-k} processors. From now on, we will be using M instead of $4^{n-k}\lceil (N+1)/4^{n-k}\rceil$ (i.e., total number of bits in each extended operand) and q instead of $\lceil (N+1)/4^{n-k}\rceil$ (i.e., number of bits from each operand stored in a single processor). According to the above discussion, the sign-extension of the numbers will give us new binary representations for the operands A and B, as follows: $A = a_{M-1}a_{M-2}\ldots a_{N-1}a_{N-2}\ldots a_1a_0$ and $B = b_{M-1}b_{M-2}\ldots b_{N-1}b_{N-2}\ldots b_1b_0$ where $a_{M-1} = a_{M-2} = \cdots = a_N = a_{N-1}$ and $b_{M-1} = b_{M-2} = \cdots = b_N = b_{N-1}$.

The M bits of each extended operand will be divided equally among the 4^{n-k} processors, such that if (x, y) are the Cartesian coordinates of a processor in the $2^{n-k} * 2^{n-k}$ grid that corresponds to a single node at level k of the fat pyramid, then the processor will contain the bits of the operands whose subscript s satisfies the condition $s \bmod 4^{n-k} = 2^{n-k}y + x$.

The algorithm is as follows: Initially, the processor at (x, y) initializes q individual carry-in terms $ca_j(x, y)$ to zero, for $0 \le j < q$; $ca_j(x, y)$ will represent the carry-in with subscript $j4^{n-k} + 2^{n-k}y + x$. It also initializes q individual propagate terms $Pr_j(x, y)$, that will correspond to groups preceding the bits of the operands it deals with, to 1. $Pr_j(x, y)$ will correspond to the propagate term with subscript $j4^{n-k} + 2^{n-k}y + x$, where $0 \le j < q$.

Then, all the processors initialize in parallel q group carry and q group propagate terms for groups of size $2^0 = 1$, as follows: $Gca_j(x,y) = GG_{i,i} = a_i b_i$ and $GPr_j(x,y) = GP_{i,i} = a_i + b_i$, for $0 \leq j < q$, where $i = j4^{n-k} + 2^{n-k}y + x$. We assume that the binary representations of the Cartesian coordinates (x,y) of each processor, in the grid of size $2^{n-k} * 2^{n-k}$ it belongs to, are as follows: $x = x_{n-k-1} \ldots x_1 x_0$ and $y = y_{n-k-1} \ldots y_1 y_0$.

Let us denote by x'_r the binary number that differs from the binary representation of x only in bit position r, that is $x'_r = x_{n-k-1} \ldots \bar{x}_r \ldots x_1 x_0$ where $0 \leq r < n-k$, and $\bar{x}_r$ represents the binary complement of x_r. In a similar way, let $y'_r = y_{n-k-1} \ldots \bar{y}_r \ldots y_1 y_0$.

Initially, pairs of processors that have the same value for the y coordinate, and whose x coordinate differs only in x_0, exchange their values of $Gca_j(x,y)$ and $GPr_j(x,y)$, where $j = 0, 1, \ldots, q-1$. We consider bidirectional communication channels in the grid of processors, which means that any two neighbor processors can send data to each other at the same time.

So, the processor at position (x,y) receives $Gca_j(x'_0,y)$ and $GPr_j(x'_0,y)$, for $0 \leq j < q$. Then, only the processors whose x_0's are 1 change their individual carry and propagate terms as follows: $ca_j(x,y) = ca_j(x,y) + Pr_j(x,y)Gca_j(x'_0,y)$ and $Pr_j(x,y) = Pr_j(x,y)GPr_j(x'_0,y)$ where $0 \leq j < q$.

The above operations update the carry-in values and the corresponding propagate terms for the bits in those processors, since for $x_0 = 1$ we have

$$Gca_j(x'_0,y) = Gc_{j',j'}, GPr_j(x'_0,y) = GP_{j',j'}, ca_j(x,y) = c_{j'+1} \text{ and } Pr_j(x,y) = P_{j'+1},$$

where $j' = j4^{n-k} + 2^{n-k}y + x - 1$. We then have $c_{j'+1} = c_{j'+1} + P_{j'+1}Gc_{j',j'}$ and $P_{j'+1} = P_{j'+1}GP_{j',j'}$ respectively, which give updated carry-in and propagate values for the corresponding bit position, after combining consecutive bits in pairs. Then, the following operation is performed in parallel in those processors that have $x_0 = 1$, for $0 \leq j < q$: $Gca_j(x,y) = Gca_j(x,y) + GPr_j(x,y)Gca_j(x'_0,y)$, otherwise (i.e., for processors that have $x_0 = 0$) $Gca_j(x,y) = Gca_j(x'_0,y) + GPr_j(x'_0,y)Gca_j(x,y)$. Also, all the processors perform in parallel the operation $GPr_j(x,y) = GPr_j(x,y)GPr_j(x'_0,y)$ for $0 \leq j < q$.

The above implies $Gc_{j',j'+1} = Gc_{j'+1,j'+1} + GP_{j'+1,j'+1}Gc_{j',j'}$ and $GP_{j',j'+1} = GP_{j'+1,j'+1}$ $GP_{j',j'}$ respectively, where $j' = j4^{n-k} + 2^{n-k}y + 2\lfloor x/2 \rfloor$. The new values of $Gca_j(x,y)$ and $GPr_j(x,y)$ will represent the group carry and the group propagate terms for pairs of consecutive bits.

Processing similar to the above continues $n-k-1$ more times. During the r^{th} execution, where $0 \leq r < n-k$, any two processors that have the same value for the y coordinate but differ in the r^{th} bit of their x coordinate (i.e., their distance is 2^r in the grid of size $2^{n-k} * 2^{n-k}$) exchange their values of $Gca_j(x,y)$ and $GPr_j(x,y)$, and then perform the following operations in parallel:

if $x_r = 1$, then for $0 \leq j < q$,

$$\begin{aligned} \{ca_j(x,y) &= ca_j(x,y) + Pr_j(x,y)Gca_j(x'_r,y), \\ Pr_j(x,y) &= Pr_j(x,y)GPr_j(x'_r,y), \\ Gca_j(x,y) &= Gca_j(x,y) + GPr_j(x,y)Gca_j(x'_r,y)\} \end{aligned}$$

otherwise

$$\{Gca_j(x,y) = Gca_j(x'_r,y) + GPr_j(x'_r,y)Gca_j(x,y)\}$$

and all the processors update the group propagate terms as follows:

$$GPr_j(x,y) = GPr_j(x,y)GPr_j(x'_r,y).$$

Just after the r^{th} execution of the above steps, where $0 \leq r < n-k$, the processor at (x,y) will contain the group carry (assuming a carry-in of 0 for the group at this point) and the group propagate terms for the q groups of 2^{r+1} consecutive bits each, that are stored in the processors with the same y coordinate and whose x coordinate in the $2^{n-k} * 2^{n-k}$ grid is in the range from $2^r\lfloor x/2^r \rfloor$ to $2^r\lfloor x/2^r \rfloor + 2^r - 1$. Also, the final value of the carry-in for the bit positions 0 through $2^r - 1$ will have been computed.

After those $n-k$ "cycles", the processor at (x,y) will contain the $2q$ auxiliary group terms for the q groups of bits $j4^{n-k} + 2^{n-k}y$ through $j4^{n-k} + 2^{n-k}(y+1) - 1$, for $0 \leq j < q$ (i.e., the combination of data contained in all the processors whose second coordinate is y). Similar types of operations are then performed for the y dimension. Groups of bits are combined using only vertical communication patterns in the grid.

So, to summarize, communications and computations occur during the first $2(n-k) = \log p$ "cycles" of the algorithm, where $p = 4^{n-k}$ is the number of processors allocated to a single node at level k of the pyramid, while only computations occur during the remaining q "cycles." The computation time of the algorithm is proportional to $\log p + q$, if we assume that the system contains m-bit processors, where $m \geq q$. If we consider 1-bit processors, then the computation time of the algorithm will be proportional to $q * \log p$, because each of the operations presented before will be executed q times in each "cycle" of the algorithm. Details and references can be found in [3].

3.1.3. Replicated Image Processing

The conventional way of mapping images onto large SIMD machines like the Connection Machine, that of assigning a processing element per pixel, tends to let major portions of the machine lie idle, especially when the picture is relatively small, as is the case in focus-of-attention image processing. We developed a method to remedy this situation, gaining on the processing time as a result. We replicate the image as many times as possible, and let each copy solve a part of the problem. The partial results from the individual copies are then aggregated to get the solution to the problem. We call this method replicated image processing. We developed replicated image algorithms for histogramming, table lookup, and convolution on the Connection Machine and compared their performance with the non-replicated algorithms for the same.

In a replicated pyramid, the cells in the hypercube are used to replicate, as many times as possible, the image stored at any given level of the pyramid. So, for example, if the image at the base (the 0-th level) of the pyramid has exactly as many nodes as there are cells in the hypercube, then only one copy of the image would be stored in the hypercube. At the first level there would be sufficient cells to store four copies of the reduced resolution image. In general, at the l-th level we would have 4^l copies of the reduced resolution image. It is

straightforward to embed the replicated reduced resolution images into the hypercube using Gray codes.

We have implemented the histogramming, table lookup, and convolution algorithms for a replicated pyramid architecture. A single level of the replicated pyramid was implemented on the Connection Machine by configuring it as an $n \times m \times m$ array of processors, where n is the number of copies of the $m \times m$ image. Communication is efficient this way, since the Connection Machine automatically assigns subhypercubes to each copy of the image. Communication within each copy remains local and is independent of the communication within other copies. Also, the logarithmic merging step, typical of the second phase of all three algorithms is very fast because corresponding elements of different copies of the image are adjacent in the Connection Machine hypercube.

A copy of the histogram of the image (in the case of the histogramming algorithm) or the entire table (in the case of the table lookup algorithm) was stored in each copy of the image. Histogramming and table lookup distribute the problem among the $4l$ copies available in such a way that each copy handles an equal number of grey-level values of the entire image. The implementations of these two algorithms are very similar, so we discuss only the histogramming algorithm. The histogramming algorithm takes advantage of the CM2 Connection Machine feature that if each processor sends a message to the processor that counts its grey-level value, then there will be as many collisions at each counting processor as the number of pixels with that grey-level value. Using the collision resolution strategy of counting all the colliding values, the histogram for the entire image can be computed in a single step. Thus, the histogramming algorithm implemented on a single level of a replicated pyramid cannot outperform a single copy histogram algorithm on the Connection Machine.

The convolution algorithm yields more interesting results on replicated pyramids. To compute a $k \times k$ convolution, k^2 copies of the image are required. (On the Connection Machine, the next power of 2 has to be chosen as the actual number of copies allocated, although only k^2 of them are used.) Each copy handles one of the kernel weights. In the first phase, each processor multiplies this kernel weight with its own grey-level value and sends the product to the appropriate processor in the same copy. In the second phase, these products are summed to obtain the final result of the convolution at every processor of every copy. We compared the time taken by the convolution algorithm implemented on the CM using a single copy against the replicated algorithm using k^2 copies. Experiments indicate that the utilization of the Connection Machine is much higher for the replicated algorithm than for the single copy algorithm. This can be explained as follows. If there are a sufficient number of physical processors to store all k^2 image copies, then the replicated algorithm performs one multiplication step (in which all k^2 kernel multiplications are performed in parallel) and $2 \log k$ addition steps to compute the multi-copy convolution. In contrast, the single copy convolution algorithm has to perform k^2 steps of multiplications and additions, all under direction of the host computer. Thus, the host overhead is much higher for the single copy algorithm than for the replicated algorithm.

There are many basic image analysis algorithms to which the methods described above can be extended in a straightforward way. For example, grey scale morphological operations are implemented in much the same way as grey scale convolution; various types of image statistics useful for image texture analysis, such as cooccurrence matrices or difference

histograms, can be computed very quickly also. Details and references can be found in [4].

3.2. Quadtrees on the Connection Machine

We developed a general technique for creating SIMD parallel algorithms on pointer-based quadtrees. It is useful for creating parallel quadtree algorithms which run in time proportional to the height of the quadtrees involved but which are independent of the number of objects (regions, points, segments, etc.) which the quadtrees represent, as well as the total number of nodes. The technique makes use of a dynamic relationship between processors and the elements of the space domain and object domain being processed.

Consider the task of constructing a quadtree for line segment data (a *PM quadtree*). In constructing a PM quadtree, a node should be assigned the color gray and subdivided if its boundary contains more than one endpoint, or if its boundary has two segments which enter it but which do not have a common endpoint within it. Initially, we have one processor allocated for the quadtree root, and one processor for each line segment, containing the coordinates of the segment's endpoints.

Consider creating an algorithm, to construct the PM quadtree for this segment data. Each segment processor initially possesses a pointer to the quadtree root processor. Each segment processor computes how many of its segment's endpoints lie within the boundary of the node to which the segment processor points; this will be 0, 1, or 2. Each segment then sends this value to the node it points to, and both the maximum and minimum of these values is computed at the node. Any node which receives a maximum value of 2 assigns itself the color gray, since this means that some single segment has both endpoints in the node's boundary. Any node which receives a maximum of 1 and a minimum of 0 also assigns itself the color gray, since this means that there are at least two segments in the node's boundary, one which passes completely through it and one which terminates within it.

Then each segment with exactly one endpoint in the node it points to sends the coordinates of that endpoint to the node. The node receives the minimal bounding box of the coordinates sent to it (this, of course, amounts simply to applying min and max operations appropriately to the coordinate components). If this minimal bounding box is larger than a point, the node assigns itself the color gray, since this means that some two segments entering the node have non-coincidental endpoints within the node.

Finally each segment with 0 endpoints in the node it points to determines whether it in fact passes through the interior of the node at all, and if so it sends the value "1" to the node, where these values are summed. If the sum received by the node is greater than 1, the node assigns itself the color gray, since this means that some two segments passing through the node do not have any endpoints in the node, which implies that they do not have a common endpoint in the node. Then all gray nodes allocate son processors. Any nodes which were not given the color gray should be colored white if no segments entered their interior (the sum is zero), and black otherwise (the sum is one).

At this point in the algorithm, we would like to have all segment processors which point to gray nodes compute which of the node's sons they belong to, and retrieve from the node the appropriate son pointer. Of course a given segment can intersect more than one of the node's sons, and we are left with the situation of wanting to assign up to four son pointers

to the segment processor's node pointer, and processing each of the corresponding sons. The solution to this dilemma is to allocate *clones* of each such segment processor, that is, to create multiple processors which represent the same segment, and all of which contain (almost) the same information. So for each segment processor pointing to a gray node, we allocate three clone processors, all of which contain the segment's endpoints and a pointer to the same node as the original segment processor. In addition, the original and its clones each contain a *clone index* from 0 to 3, with the original containing 0 and each of the clones containing a distinct index from 1 to 3. Now the original and its clones each fetch a son pointer from the node that they all point to, each one fetching according to its clone index, so that each gets a different son pointer.

The subsequent iterations of the algorithm proceed as the first, with each segment processor determining how many of its endpoints lie within the interior of the node it points to, and with the eventual computation of the colors of all the nodes on each particular level. At this point in each iteration, notice that any segment processors pointing to leaf nodes, or whose segments do not pass at all through the interior of the node to which they point, will not have any further effect of those nodes, and can thus be de-allocated and re-used later. This reclaiming of segment processors keeps the number of clones allocated for each segment from growing exponentially. In fact the number of processors required for a given segment at a given level in the construction of the quadtree will be only roughly as many as there are nodes in that level of the tree through whose interior the segment passes.

To summarize the general technique, then, we allow one processor per quadtree node, and initially allow one processor per object. Each object is given access to a sequence of shrinking nodes which contain part of it; initially all objects have access to the root node. By having each object obtain information from its node, and by combining at the node information from all of the objects who access that node, the objects make decisions about descending the quadtree from that node. For those objects which do descend, it is desirable for their various parts which lie in various quadrants of the node to descend in parallel. Thus we allow duplicate or 'clone' processors for each object, and have each processor handle just that portion of the object relevant to one quadrant of the node. Duplicate processors which determine that they can no longer effect the the node to which they point, because that node is a leaf, or because the object they represent does not overlap that node, can deactivate themselves so that they may be used later in the computations for some other object.

We see then that this technique allows us to go beyond the level of granularity of one processor for every element (space component or object) to a level where there are multiple processors for certain elements and none for others; where the processors are being used and disposed in a dynamic fashion.

The same general technique can be applied to create algorithms for several other quadtree tasks, such as shifting, rotation, and expansion, which run in time proportional to the height of the new quadtree, by computing in the parallel the rotated or expanded version of each old black leaf node, and building the new quadtree using cloning. Further details are given in [5].

3.3. Benchmarking Activities

3.3.1. Border Tracking with an Implementation on the Butterfly Parallel Computer

3.3.1.1. Background

In digital image analysis, objects visible in an image can be recognized by describing their borders, and then matching the border representations to model descriptions. There are basically two ways of tracking the borders of a given object. The first is studying local gray level values in a small neighborhood, and producing a strong response for fast gray level changes at object edges and a low response for image locations with constant or slowly varying spatial gray level distribution. The strongest edges are separated from the weaker ones by thresholding the gradient magnitude image, exploiting for example the edge strength histogram. The remaining borders are finally followed to produce coordinate lists corresponding to the 4- or 8-connected edge point patterns in the image. The other alternative differs from the first in that the segmentation phase preceding the border following is not trying to locate the edges of an object directly by studying local gray level changes, but by first locating the object body and then tracing the outer borders and the inner borders corresponding to the holes in the body. In this work an algorithm based on the second approach was developed.

The real-time operation requirement of complicated vision systems necessitates the use of fast parallel implementations of image analysis algorithms. In particular low and intermediate level operations require processing of large amounts of input data, and are potential objects for parallelization. In this study, a parallel version of the border tracking algorithm was developed for a multiprocessor system classified as a MIMD-computer. The specific computer used in testing the algorithm was the Butterfly Parallel Processor with an 88 processor configuration.

3.3.1.2. Sequential Algorithm for Border Tracking

The algorithm tracks the borders of an area-segmented binary image. The input image is processed by one pass in a row-by-row fashion, starting on the highest row and proceeding downwards. Instead of outputting edge point coordinates, a crack code representation of the border shape is used. The crack code of an object border is produced in the following way: if we follow the cracks around a border, at each move we are going either left, right, up or down; if we denote the direction 90^*i by i, these moves can be represented by a sequence of 2-bit numbers $(0, 1, 2, 3)$.

There are two kinds of borders in an image: outer and inner borders. An outer border encloses an object, whereas an inner one surrounds a hole in an object. In this algorithm the outer boundaries are followed in a clockwise order and the inner ones in a counterclockwise order. The opposite directions result from the principle of tracing the edges in such a way that the object is always kept on the right side of the trace path. The objects are taken as 4-connected and the background as 8-connected.

An image is processed by moving a 2×2 window through the image in a raster scan fashion: each row from left to right, row by row from top to bottom. The actions to be

performed depend on the neighborhood visible in the window, resulting in sixteen primary operations on the crack code strings. The primary operations resolve themselves into various subactions, such as extending a code string head or tail, merging of two strings, and creation of a new outer or inner string.

A 2×2 window doesn't always contain enough information to carry out certain string operations. An example of such a case is the presence of neighboring runs on the previous row, which often lead to the merging of two strings, or the creation of a new inner or outer string. In this algorithm a knowledge of the runs yet to come on the current row is not necessary, even if those runs turn out to be neighbors to the current run. Every decision is made using only the local history of the row and the contents of the current window. The local history of the row contains two pieces of information. The first one is a flag which indicates whether there is a neighboring run in the previous row which terminates before the current position on the row. If there is, the flag points to the right end of the neighboring run. A similar flag is maintained for the current row. When the right end of the current run is encountered, the flag is set to point to the end column if there is a neighboring run on the previous row which still continues. The presence of a neighboring run on the previous row sometimes results in the merging of two code strings or to the creation of a new inner string.

The algorithm contains two important data structures. The first is called an 'active string list' and the other one is called a 'modified string list'. While scanning a row, the active string list contains the addresses of the active strings for the current row, i.e. strings which have not yet been updated on this row. After updating a string it is moved to the modified string list. The modified string list becomes the active string list on the next row and a new empty modified list is then created. As a string becomes closed, it is removed from the lists and output.

3.3.1.3. Parallel Algorithm for Border Tracking

Parallelization of this kind of sequential algorithm seems quite straightforward. The input image is divided into a given number of equal sized blocks overlapping each other by one row, each of them is processed separately, and finally the partial results are combined. This leads to a two-stage algorithm where the second stage cannot be started before the first one is completed. The parallelization of the first part is obvious, but the necessity for parallelism in the second stage depends on the amount of partially completed data produced in the first stage. In this application the input aggregate for the merging process depends on input image size and its contents and especially on the number of image blocks. The second stage was found to be the bottleneck for the performance in several experiments, which motivated us to design a parallel algorithm for merging of partial code strings, too.

There is an important point to be considered with the partition of image data, though. The easiest way of doing the partitioning is to assign an equal number of rows to each processor. This is the first approximation of equal work load for the processors. But a simple principle like this often leads to improper balancing among the processors if the contents of the image is not evenly distributed. Because the execution time of a parallel program step is dominated by the slowest parallel subtask, it is important to do the division of input data by exploiting a better estimation for the quantity of work.

The main modification to the sequential algorithm is due to the possibility that an object border may traverse several image blocks. For this reason, the top row and bottom row of a block contribute to extra actions in many of the primary cases. To overcome these implications a few additional members must be included in the code string data structure. The most important of these are binary flags to indicate an instant of a lower or higher border crossing for string head and tail components. Also the intersection coordinates must be recorded in the structure. These additional data are used in the merging stage for deciding on connectedness of partial strings in neighboring image blocks.

Classification of strings into outer and inner borders becomes somewhat meaningless within an image block if they traverse two or more partitions, because just by looking at a partial string it is not possible to make the distinction. These attributes are useless in the merging stage, too. If a string can be completed in a block, though, its classification holds. There is a simple way to determine reliably whether a completed string is an outer one or an inner one. This can be carried out by studying the local neighborhood of the highest border point of the string, called the reference point of the string. It is the first point of the associated border that is found during scanning the image. The special meaning of reference point is due to the action of creating a new partial string at this location. Because borders are always followed the same direction keeping the object on right the crack codes emanating from the reference point are different with an outer string and an inner string. The crack code sequence for an outer string in the vicinity of the reference point, coming along the head and continuing along the tail, is '...1-0...'. The sequence for an inner string is respectively '...2-3...'. During the border tracking stage, each processor assigns a reference point to every string in its block. If a string crosses the upper limit of the block, the intersection point is also the local reference point of the partial string. After the merging step, the global reference point is searched for along the combined string chain and the classification is finally done.

Conclusions

A general algorithm for extracting object borders was developed. The binary image is processed in a raster scan fashion in one pass, producing crack code strings describing the borders. The objects are regarded as 4-connected, but an 8-connected version is easily achieved. The algorithm can be generalized to process gray level images, too.

A two-stage parallel version of the algorithm was developed, where the input image is partitioned into partially overlapping equal-sized blocks, each of which is processed by a separate processor. The first step produces crack code descriptions of the object borders hitting the blocks, in parallel. The second step is required for merging the incomplete border strings traversing more than one image block.

Our experiments on the Butterfly Parallel Processor reveal that with an image size of 512 by 512, the speedup increases nearly linearly up to the point where about twenty processors are in use. After that, the contention for shared memory resources starts leveling off the performance growth severely. One important source for the increase in execution time after this critical point is the merging step which is more and more heavily loaded as the number of incomplete border descriptions increases with the increased number of image partitions.

The merging step can be omitted from the algorithm. Additional details and references can be found in [6].

3.3.2. Parallel Matching of Attributed Relational Graphs

3.3.2.1. Background

In many computer vision applications, it is necessary to utilize structural analysis in pattern classification. To facilitate this, an appropriate structural description of objects and their mutual relations is required. In addition, a fast classifier for these descriptions should be available.

Attributed relational graph description has been found to be very suitable for computer vision. There are several approaches to match the graphs. These algorithms vary from complex syntactic methods to simple matching techniques. Unfortunately, the optimal comparison of two graphs is inherently an NP-complete problem. Therefore, the computational efforts required increase exponentially with the number of nodes in the graphs. For example, the matching of two k-node graphs using conventional algorithms may require the construction and evaluation of $k! * k!$ solution candidates in the worst case. This means that when using conventional computers and algorithms the optimal comparison of graphs with more than a few nodes may take too long for practical applications.

In most applications, however, some physical and heuristic constraints can be applied to slow down the exponential explosion. Furthermore, the matching can be parallelized to achieve almost a linear speedup in a multi-processor environment. Finally, the entire structural classifier, based on graph matching, can be parallelized to match the graph to all the models simultaneously.

We describe a fast graph matching algorithm and its parallel implementation on two MIMD computers: a Butterfly Parallel Processor, and a Transputer system.

3.3.2.2. A Fast Graph Matching Algorithm

In our earlier work, a fast matching algorithm was developed for structural classification. The algorithm can be used to solve both isomorphic and monomorphic problems. The resulting numeric value describes the edit distance between the graphs. The algorithm has been proved to be useful for defect classification in visual inspection applications.

The edit distance is defined as the minimum number of changes to make the graphs similar to each other. The changes allowed are the deletion and the addition of a node, a link, or an attribute. The structural classifier calculates the edit distance between the graph to be classified and the model graphs representing different classes. The graph is classified in the class for which the minimum distance was found.

The basic problem is to find the best possible mapping between the nodes of the graphs to be matched. To accomplish this, a depth-first search is used to build a state space tree of solution candidates. When the search terminates, the resulting candidates are described by paths from the root to the leaves. In the worst case, the tree contains all the combinations

of the two sets of nodes corresponding to the graphs. For this reason, heuristic constraints are applied to pruning the tree in order to achieve sufficient speed.

The first heuristics for limiting the size of the tree is using a certain predetermined model node to align the matching process. This base node represents the most important object or part of the object in the model description. If the problem graph contains the base node, a partial match of the graphs has been found, and only one out of k main branches of the tree remains for further study. The second heuristics exploits the local structural similarities of the graphs. The remaining $k-1$ nodes are matched level by level in the graphs starting on the successor levels of the base nodes. The resulting state space tree often consists of only a few paths.

This ordered matching enables a speedup of several orders of magnitude without sacrificing recognition accuracy, because all feasible mappings between the graphs are still considered.

The best match is found by evaluating each candidate and choosing the one which represents the minimum edit distance. In the case of monomorphism, empty (NIL) nodes are ignored in the evaluation. The evaluation time is minimized by using lookup tables for edit distances between node and link pairs.

3.3.2.3. Implemented Algorithm

The original algorithm was implemented on a Symbolics 3645 which supports list processing by hardware. MIMD-classified multiprocessing systems usually have no special processors tailored for symbolic data processing but are rather designed for numerical calculations. To better utilize the computational power of our target computers, all the symbols (node attributes, link attributes etc.) are hashed into integer values. The method for constructing the state space in a depth-first manner is changed to a breadth-first type algorithm, and the evaluation process is sped up by including several auxiliary tables in the algorithm.

3.3.2.4. A Parallel Algorithm

For a given number of processors in a MIMD computer, the most effective parallelization is reached by parallelizing the outermost loop in the program. In this way, the speedup is nearly a linear function of the number of processors if there is no interaction between the loops. The degree of sublinearity of a speedup curve is directly affected by the communication overhead between parallel processes. The generation of a state space can be distributed effectively to processors in such a way that every processor constructs its own main branch of the tree, which is the outermost loop in the sequential program. This gives an asymptotic speedup of $O(k)$, where k stands for the size of the model graph counted in nodes. More speed can be achieved by parallelizing the process deeper in the tree. For example, if parallelization is done one level deeper there will be $k-1$ processors for each main branch. Thus the speedup is $O(k*(k-1)) = O(k*k)$. This has the side effect that the number of processors increases very quickly as the graph size is increased.

For practical applications, a compromise between speed and system size has to be made. In this work, the process of state space generation is parallelized on the highest level of the

tree, which yields a speedup of $O(k)$. Identification of the base node in the problem graph reduces the size of the tree to one main branch, in which case the parallelization is carried out on the next highest level. The speedup here is $O(k-1) = O(k)$ compared to the respective sequential version.

Parallelization of the state space evaluation is done in the same way by dividing the tree into a given number of equal-sized subtrees.

3.3.2.5. Results

The parallelization of the algorithm was designed especially to suit MIMD computers, like the Butterfly Parallel Processor and the Transputer network used in our experiments. The Transputer-based multiprocessor system was used to study the properties of our algorithm. Butterfly experiments were also performed to compare the performance of our system to a commercially available multiprocessor. The results show that the program runs two to three times faster in the Transputer system than in the BPP. The performance ratio, however, strongly depends on the system clock frequencies and the speed of the IC components selected for the hardware implementation of these target computers. Programming languages were also different (C with the BPP and Occam with Transputers), which contributes to comparison difficulties.

With the graph sizes of three to seven, the speedup of the parallel algorithm is nearly linear. The increasing deviation from a linear speedup with increasing graph size results from the more extensive communication overhead, due to longer messages describing the properties of the graphs. With larger graph sizes, our Transputer system runs out of processors if no modifications to the algorithm are done. If there are M branches to be processed with the maximum of N processors, the generation and the evaluation of the state space must be done in $L = \lceil M/N \rceil$ consecutive steps. The execution time will thus be $L * T$, where T is the processing time for N branches. Details and references can be found in [7].

4. Parallel Iterative A* Search

4.1. Overview

In this work, we developed a distributed best-first heuristic search algorithm, *Parallel Iterative A** (PIA^*). We show that the algorithm is admissible, and we give an informal analysis of its load balancing, scalability and speedup. To empirically test the PIA^* algorithm, a flow-shop scheduling problem has been implemented on the BBN Butterfly Multicomputer using up to 80 processors. From our experiments, the algorithm is capable of achieving almost linear speedup on a large number of processors with relatively small problem size.

4.2. The A* Algorithm

The A^* search procedure can be described using graph-theoretical terms. For any problem instance, a state-space graph is implicitly defined. Each node in the state-space graph represents a state. As A^* proceeds from a start node s, nodes in the state-space graph are gradually expanded by a node expansion operator.

A^* finds an optimal cost path from the start node, s, to a set of goal nodes. In A^*, a node n is assigned an *additive cost*, $f(n) = g(\ ) + h(n)$, where $g(n)$ is the actual cost of reaching node n from s, and $h(n)$ is the heuristic or estimated cost of reaching a goal node from node n. A variant of the A^* search procedure which does not test for duplicate nodes is provided below:

1. Put the start node s on a list called OPEN.
2. If OPEN is empty, exit with failure.
3. Remove from OPEN a node n whose f value is minimum.
4. If n is a goal node, exit successfully with a solution.
5. Expand node n, generating all successors that are not ancestors of n and adding them to OPEN.
6. Go to step 2.

The omission of the check for duplicate nodes is appropriate if the state-space graph is a tree. For a graph search, there is a tradeoff between the computational cost of testing for duplicate nodes and that of generating a larger search tree. The computation required for identifying duplicate nodes can be quite substantial, especially in a distributed computational environment. In the following discussion, we assume it is worthwhile to omit the test for duplicate nodes.

4.3. The PIA* Algorithm

PIA^* proceeds by repetitive synchronized iterations. At each iteration, processors are synchronized twice to carry out two different procedures: the *node expansion procedure* and the *node transfer procedure*. Operations are largely local to the processor in the node expansion procedure and are completely local in the node transfer procedure. Data structures in PIA^* are distributed to avoid bottlenecks. Node selection, node expansion, node ordering and successor distribution operations are fully parallelized. Processors performing searches which are not following the current best heuristics are synchronized to stop as soon as possible to reduce search overhead (the increase in the number of nodes that must be expanded owing to the introduction of parallelism). During processor synchronization, speculative computations are contingently performed at each processor, trying to keep processors always productively busy, to reduce synchronization overhead. Unnecessary communications are avoided as long as processors are performing worthwhile search to reduce communication overhead. A symmetric successor node distribution method is used to control load balancing. Finally, the correct termination of PIA^* is established by its iterative structure.

The parallel architecture model we assume consists of a set of processor-memory pairs which communicate through an unspecified communication channel. The communication channel can be realized using a shared memory or by message passing. Memory referencing through local memory is completed in constant unit time. A remote reference through the communication channel, however, requires $O(\log P)$ time in the worst case with no conflicts, where P is the number of processors. Note that this architecture model is general enough to subsume most scalable multicomputers which are currently available commercially.

4.3.1. Data Structures

In each processor j, two lists are maintained in its local memory: the work list (WL_j) and the reception list (RL_j). WL_j is a priority queue and RL_j is a simple list. At the beginning of each iteration, WL_j contains the sorted nodes awaiting expansion by processor j, and $RL_j = \emptyset$. During the node expansion procedure, successor nodes generated are distributed to RL_j, $j = 0 \cdots P-1$, using a successor distribution algorithm to be described below.

Henceforth, we will use WL_j^i to denote the set of nodes in the work list of processor j at the beginning of iteration i. If the subscript is omitted, it means the union of all P processors' work lists. That is,

$$WL^i = \bigcup_{j=0,\ldots,P-1} WL_j^i.$$

If the superscript is omitted, it means for all iterations. Similar notation will be used throughout this section.

Initially, $WL^0 = s$ and $RL^0 = \emptyset$. As PIA^* proceeds, WL^i is similar to a snapshot of the OPEN list in A*, but distributed.

4.3.2. Iteration Threshold, Mandatory Nodes and Speculative Nodes

A threshold t^i is associated with each iteration i, with $t^0 = h(s)$. At each iteration, *all* nodes in WL^i with $f(n) \leq t^i$ will be expanded, and *some* nodes with $f(n) > t^i$ will be expanded.

A node $n \in WL^i$ is thus called a *mandatory node* if $f(n) \leq t^i$. Let M^i, which is a subset of WL^i, be the set of all mandatory nodes. As we shall see later, all mandatory nodes will eventually be selected for expansion by either A^* or PIA^* in the worst case.

A node $n \in WL^i$ is defined to be a *speculative node* if $f(n) > t^i$. Let S^i be the set of all speculative nodes. Then, $WL^i = M^i \cup S^i$.

Initially, $M^0 = s$ and $S^0 = \emptyset$.

4.3.3. The Node Expansion Procedure

The node expansion procedure at each iteration i operates as follows. A processor j first expands all the nodes from M_j^i and, by an algorithm to be described below, puts all the successor nodes generated into the reception lists (RL). Then, as long as any other processor is expanding a mandatory node, this processor continues to expand the *best* speculative node from S_j^i. When all the nodes from M^i have been expanded, all processors synchronize for the node transfer procedure which is described below.

In PIA^*, successors generated by a node expansion are not considered for expansion until the next iteration; they are added to RL immediately after they are generated

4.3.4. Determining $t^{(i)}$

The threshold for the discrimination of mandatory nodes and speculative nodes at iteration $i+1$, $t(i+1)$, is defined as the maximum of:

(a) t^i, and

(b) the minimum cost of:

(b_i) all successors generated from nodes in M^i, and

(b_{ii}) the nodes in S^i.

There is a simple, efficient parallel method for computing t^{i+1}. For each processor j, a local constant c_j, which is the minimum cost of (a) all successors generated from nodes in M_j^i, and (b) the best node in S_j^i, can be computed during the node expansion procedure. Then, the minimum of c_j, $j = 0 \cdots P-1$, can be computed in parallel in time $O(\log P)$ while processors are synchronized; t^{i+1} is then the maximum of t^i and the computed minimum.

4.3.5. The Successor Distribution Algorithm

Successor nodes generated by a processor are put into RL_j, $j = 0 \cdots P-1$, in a *multiple round-robin* fashion. More precisely, suppose that the most recent successor node generated by processor j is added to RL_i. Then the next successor node generated by processor j will be added to RL_k, where $k = (i+1 \pmod P)$. At each iteration, processor j sends its first generated successor to RL_j.

The advantage of this approach is that it is simple to implement and its symmetric structure helps PIA^* attain the desired load balancing. Since the successors generated are

not considered for expansion until the next iteration, some optimization can be made for message-passing architectures. Messages for successor distribution can be asynchronous so that computation and communication can be overlapped. For architectures which require large communication setup time, successor nodes generated can be distributed and cached in local memory and not sent until an efficient message size for the underlying architecture is reached.

4.3.6. The Node Transfer Procedure

After the node expansion procedure, each processor j empties the nodes from RL_j and inserts them into WL_j to form a new priority queue for the next iteration. Note that the node transfer procedure is completely local; no communications between processors are required.

4.3.7. Termination

When a mandatory node is found to be a goal node by a processor, a message can be broadcast to inform all processors to terminate. If a speculative node is found to be a goal node, this node is simply added to RL because it may not be an optimal goal node.

PIA^* can terminate, failing to reach a goal node, when $WL^i = \emptyset$. $WL^i = \emptyset$ if and only if $WL^i_j = \emptyset$, for all $j = 0 \cdots P-1$. This state can be recognized and broadcast to all processors at the end of the node transfer procedure.

4.4. Analysis

The algorithm can be proven to be *admissible*, meaning that if a goal state is reachable then an optimal goal will be reached. Informal analyses and experiments have shown that the algorithm maintains good load balancing among the processors and can achieve a nearly linear parallel speedup. Details and references can be found in [8].

5. Parallel Matrix Operators

5.1. Generalized Matrix Inversion on the Connection Machine

This work is concerned with the practical implementation of algorithms for generalized inversion (g-inversion) of a matrix on the Connection Machine.

The Connection Machine is extremely well suited to data level parallelism. Accordingly the suitable algorithms are those which are deterministic and exhibit massive data parallelism. While there are several numerical algorithms available for matrix inversion, we recommend the Ben-Israel-Greville algorithm as it has the following distinct advantages:

i. Deterministic

The procedure involves only matrix multiplications and the initial approximation can be chosen to ensure convergence deterministically.

ii. Reliable

If the matrix is singular or rectangular, the least-squares or Moore-Penrose inverse is obtained, so the process does not fail.

iii. Stable

The algorithm is self-correcting and stable, and permits the use of coarse precision at earlier stages and finer precision towards the end.

iv. Linear time

Since the algorithm is entirely based upon repetitive parallel matrix multiplication (which takes linear time), it generates the inverse in linear time.

v. Scalable

The basic steps in this algorithm allow extension to larger matrices by using the virtual processor configuration on the Connection Machine. Here each physical processor can simulate a two-dimensional grid of virtual processors. In such a case, the speed of each processor is reduced by a factor of V/P, where

V = total number of virtual processors

and P = total number of physical processors.

(Details and reports of experiments with random matrices can be found in [9].)

The complexity of matrix-partitioning schemes for the g-inversion on the Connection machine was analyzed. It turns out that the use of the virtual processor configuration on the Connection Machine is of comparable efficiency to using any partitioning scheme, when the multiplicative iterative scheme is used for g-inversion. (See [10].)

5.2. Grid Evaluation-Interpolation on the Connection Machine

5.2.1. Grid Evaluation-Interpolation using Tensor Products and General Inversion

The interpolation and approximation of functions of two or more independent variables have recently become important because of their extensive technical applications in a wide range of

fields—digital image processing, digital filter design, topography, photogrammetry, geodesy, and optical flow, to mention only a few. In all these applications it is required to construct formulae that can be efficiently evaluated.

Tensor products are widely used in the evaluation and interpretation of functions as well as 2D and 3D image blocks. We implemented a tensor product on the Connection Machine. We developed a set of ready to use tensor product approximation schemes for data arrays of size (2×2), (3×3) and (4×4) in 2D and 3D. We use bilinear, trilinear and higher order forms for this purpose. These schemes can be used recursively for larger array sizes on the Connection Machine. The tensor product approximation is computationally economical to implement. For instance, in the 3D case for the $(n \times n)$ grid, we need to precompute and store only the inverses of three matrices of size $(n \times n)$ rather than one of size $(n^3 \times n^3)$. Obviously, error is introduced in such an approximation; Gordon and Schumaker provide explicit error bounds for this, as well as other related approximation schemes. Details and numerical examples are given in [11].

5.2.2. Grid Evaluation-Interpolation using Multivariable Spline-Blending Approximation

Instead of using polynomials $(1, x, x^2, \ldots)$, one can use Lagrange or spline functions for interpolation and use blending approximation that combines the approximations obtained using coarse and fine grids. Computationally, one of the most convenient classes of methods for this purpose is the "product-operator method," where the approximating function is calculated by treating the individual variables separately. A particular scheme is the projection operator technique of Gordon, which uses the cardinal splines, and provides a substantial saving in the number of function values that are required to approximate a function to a prescribed accuracy.

We developed a Connection Machine implementation of the projection operator technique for multivariable cardinal spline interpolation. For this purpose we implemented several data-parallel operations such as inner product and tensor product of vectors (whose components are single variable polynomials). These give rise to the functional form of approximation: hence we can perform symbolic differentiation and integration of the approximating functions directly. The technique uses orthogonal polynomial basis functions and their tensor products without requiring matrix inversion. Details and numerical examples can be found in [12].

6. Conclusions and Future Research

During the remaining two years of the contract we will focus our attention on the RAMBO project. We hope to have an initial version of the RAMBO system competed by the end of 1989 that will be capable of illuminating the target sensors when the target is undergoing a relatively simple motion, such as pure translation or pure rotation. Additionally, we plan to study the following specific problems during the coming year:

1. target reacquisition. It is possible that the target might move completely out of RAMBO's field of view either due to unexpected changes in the target's motion, or errors in estimation of the target's motion. We plan to study visual search strategies, based on both explicit models for the allowable motions of the target and estimates of the accuracy with which the parameters of those models can be determined, that will allow RAMBO to reacquire the target if it is lost from the field of view.

2. task planning. Our current task planning system is very simple, employing a greedy like algorithm to choose the next subtask to achieve. It does not include in its plan formation considerations of escape if the target motion changes or visibility constraints for monitoring both the motion of the target or the execution of the plan. We are designing a more general plan generation and monitoring system that makes extensive use of predetermined possible paths for subtask execution, and can generate plans for more complex task specifications.

3. The class of tasks that RAMBO can achieve depends on the accuracy with which it can estimate the motion of the target. We are studying the use of Kalman filtering methods to improve the accuracy of both pose estimation and motion estimation.

Other projects that we have initiated include development of methods for stabilizing image sequences for tele-operation, methods for performing texture analysis on range data and more goal directed parallel object recognition and pose estimation algorithms.

References

[1] D. DeMenthon, "Reconstruction of a Road by Matching Edge Points in the Road Image," June 1988, CAR-TR-368, CS-TR-2055, June 1988.

ABSTRACT: A method for the reconstruction of a road in 3D space from a single image is presented. The world road is modelled as a space ribbon generated by a centerline spine and horizontal *cross-segments* of constant length (the *road width*) cutting the spine at their midpoints and normal to the spine. The tangents to the road edges at the end points of cross-segments are also assumed to be approximately parallel. These added constraints are used to find pairs of points (*matching points*) which are images of the end points of world cross-segments. Given a point on one road image edge, the proposed method finds the matching point(s) on the other road image edge. Surprisingly, for images of road turns, a point on one road image edge has generally more than one matching point on the other edge. The extra points belong to "ghost roads" whose images are tangent to the given road image at these matching points.

Once pairs of matching points are found in the image, the reconstruction of the corresponding world cross-segments is straightforward since cross-segments are assumed to be horizontal and to have a known length. Ghost road cross-segments are discarded by a dynamic programming technique. A benchmark using synthetic roads is applied, and the sensitivity of the road reconstruction to variations in width and bank of the actual world road is evaluated and compared to the sensitivity of two other algorithms. Experiments with a sequence of actual road images as the Autonomous Land Vehicle (ALV) moves down a road are also presented.

[2] C.A. Sher and A. Rosenfeld, "A Pyramid Hough Transform on the Connection Machine," CAR-TR-421, CS-TR-2182, January 1989.

ABSTRACT: A pyramid programming environment on the Connection Machine is presented. The mapping between the Connection machine and pyramid structures is based on a scheme called Shuffled 2D Gray Codes. A pyramid Hough transform, based on computing the distances between line or edge segments and enforcing merge and select strategies among them, is implemented using this programming environment.

[3] S. Ziavras and L.S. Davis, " Fast Addition on the Fat Pyramid and its Simulation on the Connection Machine," CAR-TR-383, CS-TR-2093, August 1988.

ABSTRACT: This paper presents an algorithm for fast addition on the fat pyramid. The fat pyramid is a pyramid in which the storage space and the processing power allocated to a single node increase as the root of the pyramid is approached. The addition algorithm is based on a carry-lookahead technique. The computation time of the algorithm is proportional to $\log p + q$ for operands of size $p * q$ bits, when p processors are used to deal with the numbers. The addition algorithm was simulated on the Connection Machine; some performance results are presented in this paper.

[4] L.S. Davis and P.J. Narayanan, "Efficient Multiresolution Image Processing on Hypercube Connected SIMD Machines," CAR-TR-430, CS-TR-2227, April 1989.

ABSTRACT: We describe two approaches to efficiently processing small images on hypercube connected SIMD machines. The first approach, called fat images, is based on distributing the bits representing the grey level (or other feature) from each pixel across the processors of a sub-hypercube, using Gray coding techniques to obtain a good mapping of the fat image into the hypercube. The second method, called replicated images, involves generating as many copies of the small image as will fit into the machine, and then distributing the computation of basic image processing operations across the copies. For the replicated images, we present algorithms for histogramming, table lookup and convolution, and describe the results of implementing the convolution algorithm on a 16K processor Connection Machine II.

[5] T. Bestul, "A General Technique for Creating SIMD Algorithms on Parallel Pointer-Based Quadtrees," CAR-TR-420, CS-TR-2181, January 1989.

ABSTRACT: This paper presents a general technique for creating SIMD parallel algorithms on pointer-based quadtrees. It is useful for creating parallel quadtree algorithms that run in time proportional to the height of the quadtrees involved but that are independent of the number of objects (regions, points, segments, etc.) which the quadtrees represent. The technique makes use of a dynamic relationship between processors and the elements of the space and object domains being processed.

[6] T. Seppänen and K. Pehkonen, "A Generalized Algorithm for Border Tracking with an Implementation on the Butterfly Parallel Processor," CAR-TR-388, CS-TR-2099, August 1988.

ABSTRACT: This report describes a generalized one-pass algorithm for border tracking of objects in thresholded binary images. The input image is scanned from top to bottom, from left to right. On each row, partial border descriptions produced on previous rows are updated according to run ends on the current row. Borders are represented by crack code strings following the outer borders in a clockwise direction, and the inner borders in a counterclockwise direction.

Secondly, a parallelized version of the algorithm is presented with an implementation on a Butterfly Parallel Processor. The program was developed based on the Uniform System approach which supports a shared memory model of computation. The input image is partitioned into equal sized blocks, and each partition is assigned to a separate processor. Partially completed border descriptions gathered from the blocks are finally merged in parallel.

[7] T. Seppänen, T. Westman and M. Pietikäinen, "Parallel Matching of Attributed Relational Graphs," CAR-TR-376, CS-TR-2073, July 1988.

ABSTRACT: This report presents experiments with a parallel algorithm for matching attributed relational graphs. The algorithm generates a state space tree in a breadth-first manner and then evaluates the tree by computing the edit distance for each candidate solution. The parallelization method used is best suited for MIMD-type computers. The first target machine is the Butterfly Parallel Processor, in which the programs were developed on Uniform System software supporting a shared memory model of computation. The second multiprocessor is a link-oriented Transputer-based system. In this

system, concurrent processes communicate through message channels. The experiments show that nearly linear speedup can be achieved by parallelizing the algorithm in the outermost loop.

[8] S. Huang and L.S. Davis, "Parallel Iterative A* Search: An Admissible Distributed Heuristic Search Algorithm," CAR-TR-434, CS-TR-2233, April 1989.

ABSTRACT: In this paper, a distributed heuristic search algorithm is presented. We prove that the algorithm is admissible and give an informal analysis of its load balancing, scalability, and speedup. A flow-shop scheduling problem has been implemented on a BBN Butterfly Multicomputer using up to 80 processors to empirically test this algorithm. From our experiments, this algorithm is capable of achieving almost linear speedup on a large number of processors with relatively small problem size.

[9] E.V. Krishnamurthy and S.G. Ziavras, "Matrix g-Inversion on the Connection Machine," CAR-TR-399, CS-TR-2125, October 1988.

ABSTRACT: The generalized inversion of a matrix has many applications. This report considers the implementation of the Ben-Israel-Greville algorithm for finding the Moore-Penrose inverse of a matrix. This algorithm is highly suitable for data-level parallelism and has several advantages: linearity, stability, reliability, determinism and scalability. Connection Machine experiments with random matrices of different dimensions are reported.

[10] E.V. Krishnamurthy and S.G. Ziavras, "Complexity of Matrix Partitioning Schemes for g-Inversion on the Connection Machine," CAR-TR-400, CS-TR-2126, October 1988.

ABSTRACT: Theoretical results concerning partitioning of large matrices for g-inversion are outlined. The complexity and performance analysis of these methods on the Connection Machine are described. It turns out that the use of the virtual processor configuration on the Connection Machine is of comparable efficiency to using any partitioning scheme, when the multiplicative iterative scheme is used for g-inversion.

[11] E.V. Krishnamurthy and S.G. Ziavras, "Grid Evaluation-Interpolation on the Connection Machine using Tensor Products and g-Inversion," CAR-TR-401, CS-TR-2127, October 1988.

ABSTRACT: Tensor products are widely used in the evaluation and interpolation of functions as well as 2D and 3D image blocks. This report describes the implementation of the tensor product method on the Connection Machine and its applications.

[12] E.V. Krishnamurthy and S.G. Ziavras, "Multivariate Spline-Blending Approximation on the Connection Machine," CAR-TR-402, CS-TR-2132, October 1988.

ABSTRACT: This report describes the principles of the projection operator technique for multivariable cardinal spline-blending approximation on the Connection Machine. This technique requires data-parallel operations for polynomial (single and multivariable) evaluation and hence is best suited for implementation on the Connection Machine. The basic operations needed are the inner product and the tensor product of vectors

whose components are polynomials or their evaluated values. The spline-blending approximation has several applications: finite-element methods, digital image processing, optical flow and topography.

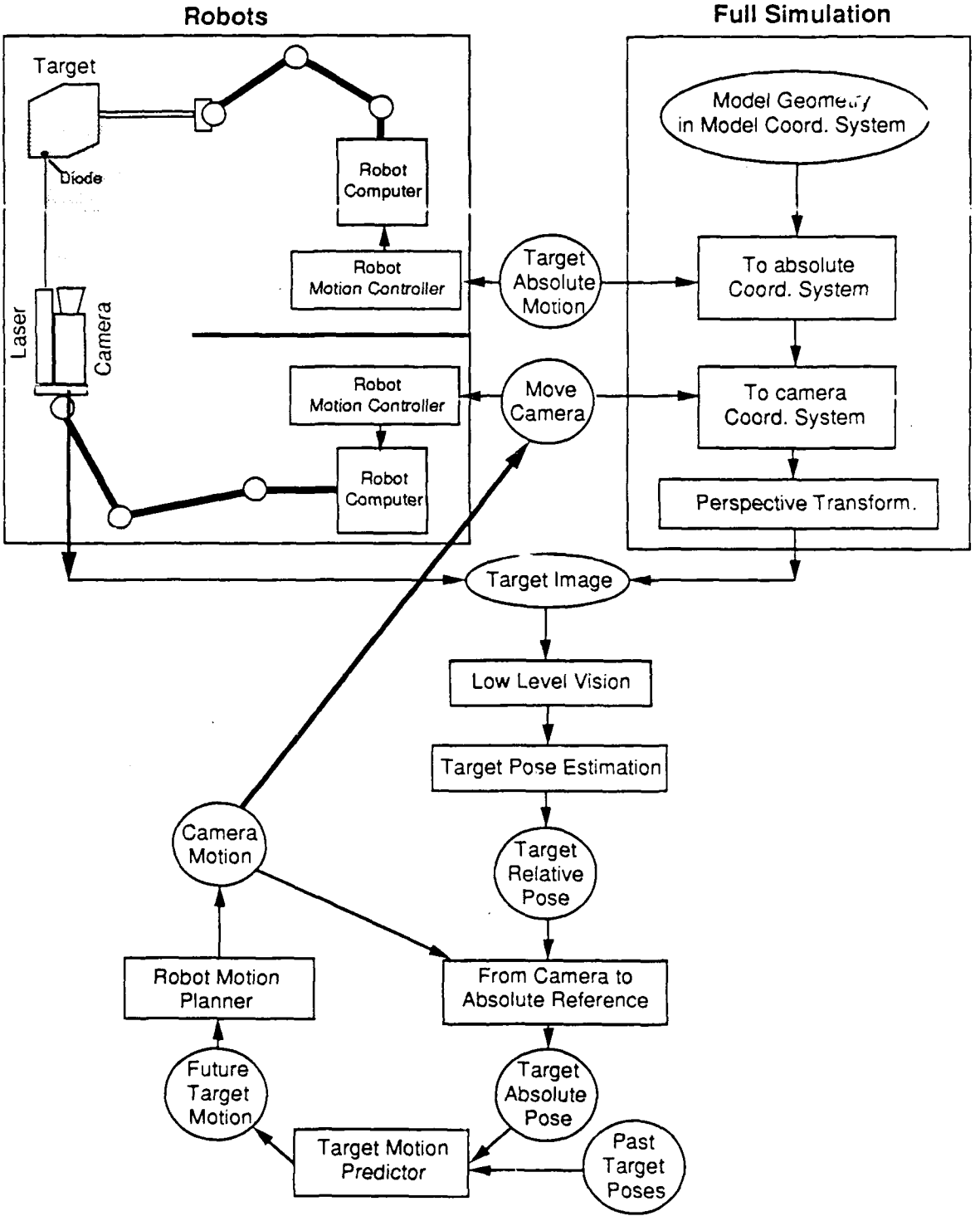


Figure 1. Vision-based control loop for a robot acting on a moving body.

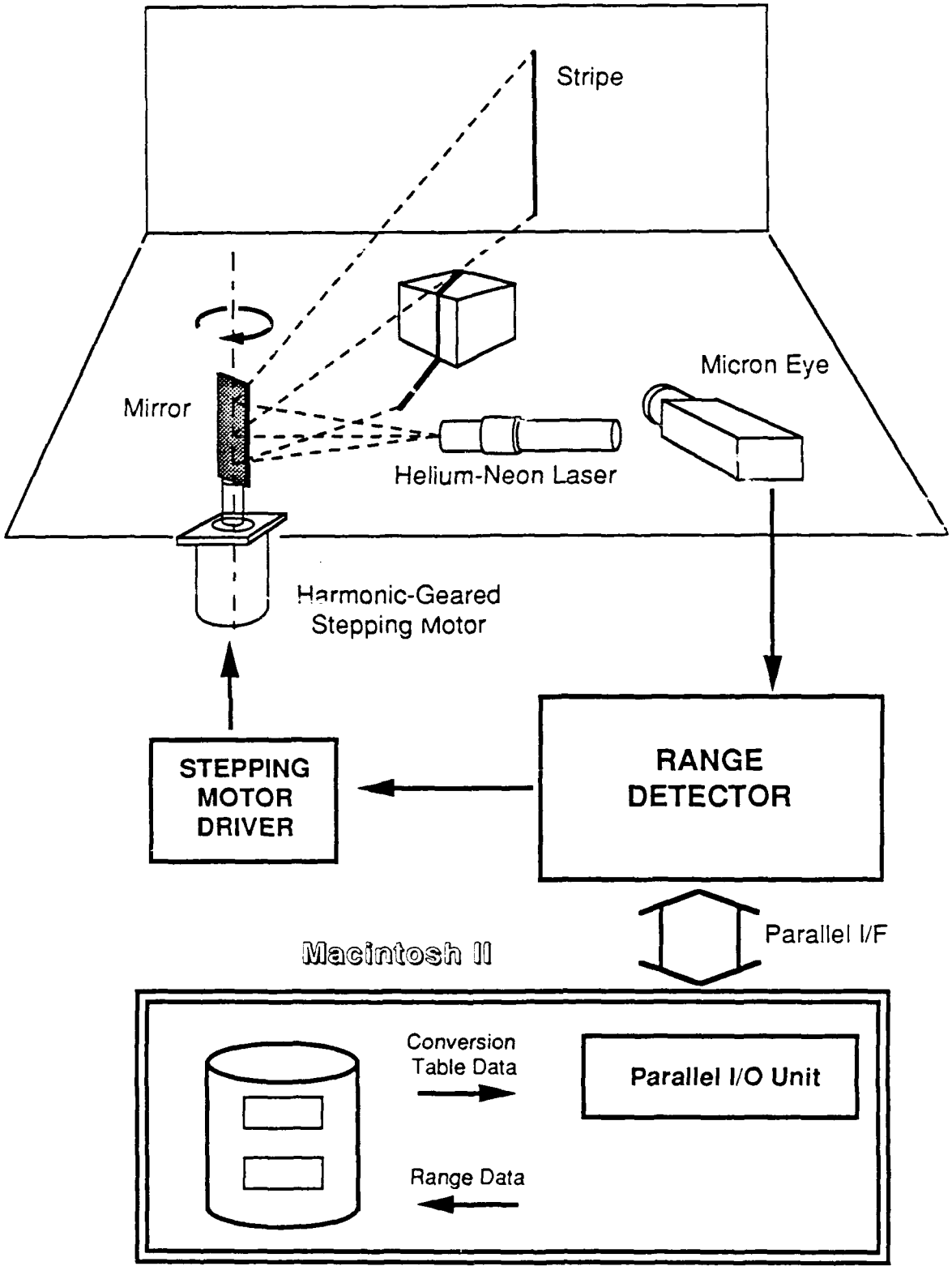


Figure 2. Range Scanner System

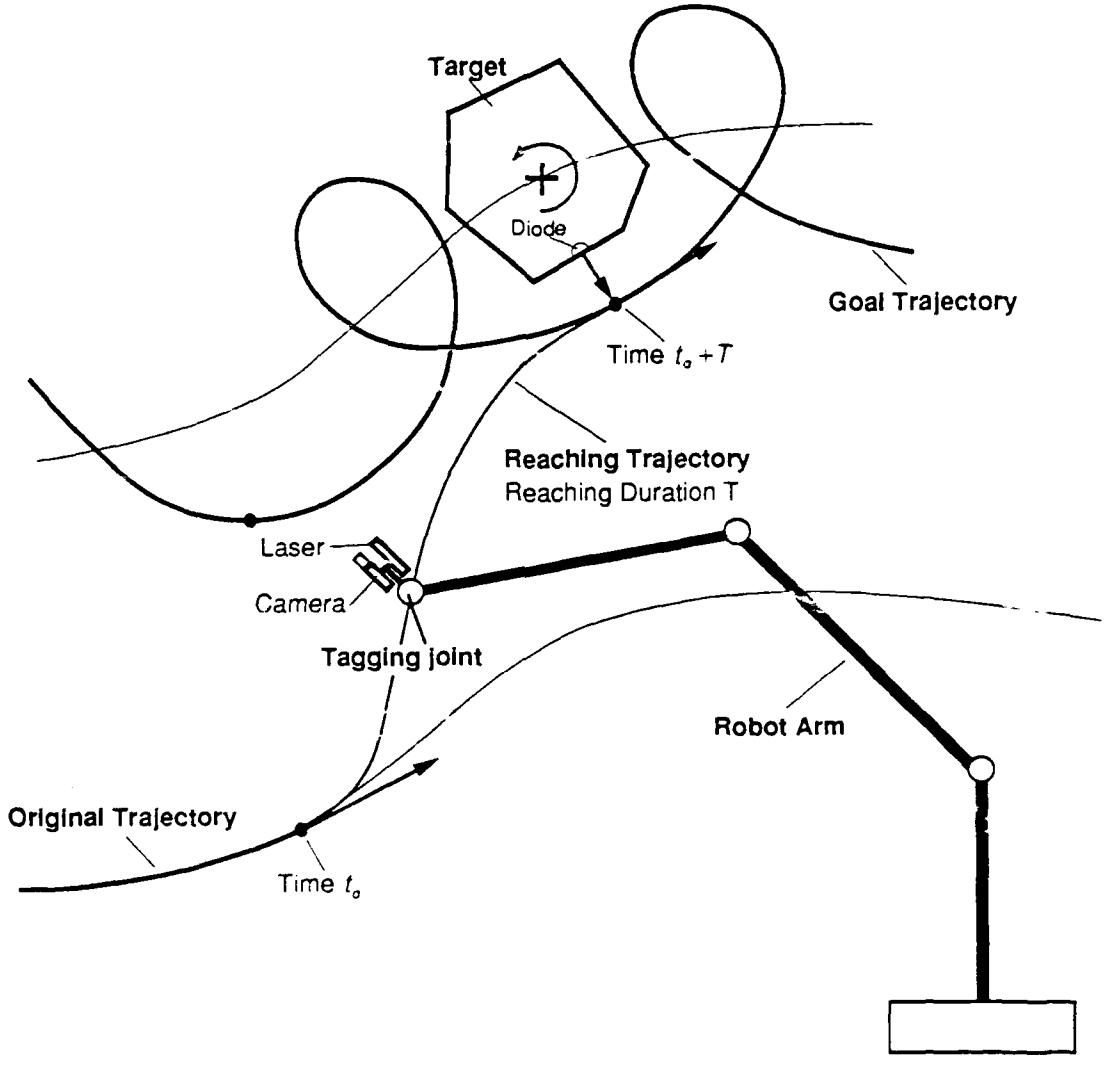


Figure 3. A tagging joint takes a reaching trajectory during time T to reach a goal trajectory required to grip a handle on the target.

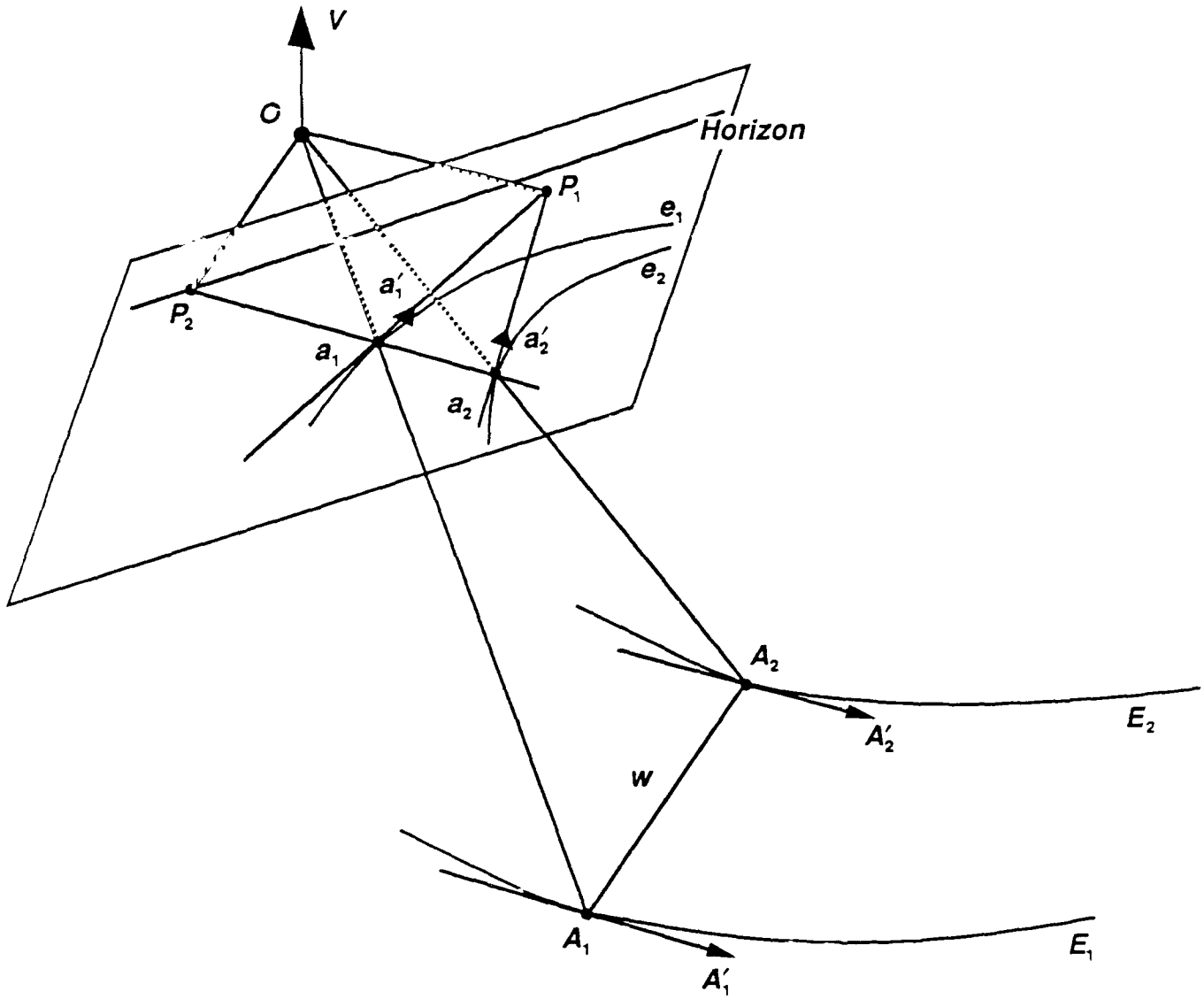


Figure 4. The cross-segment of the world road is assumed horizontal and perpendicular to the tangents at its end points. The tangents are assumed parallel. A condition satisfied by the matching points in the image which also involves the image tangents and the vertical direction is deduced.

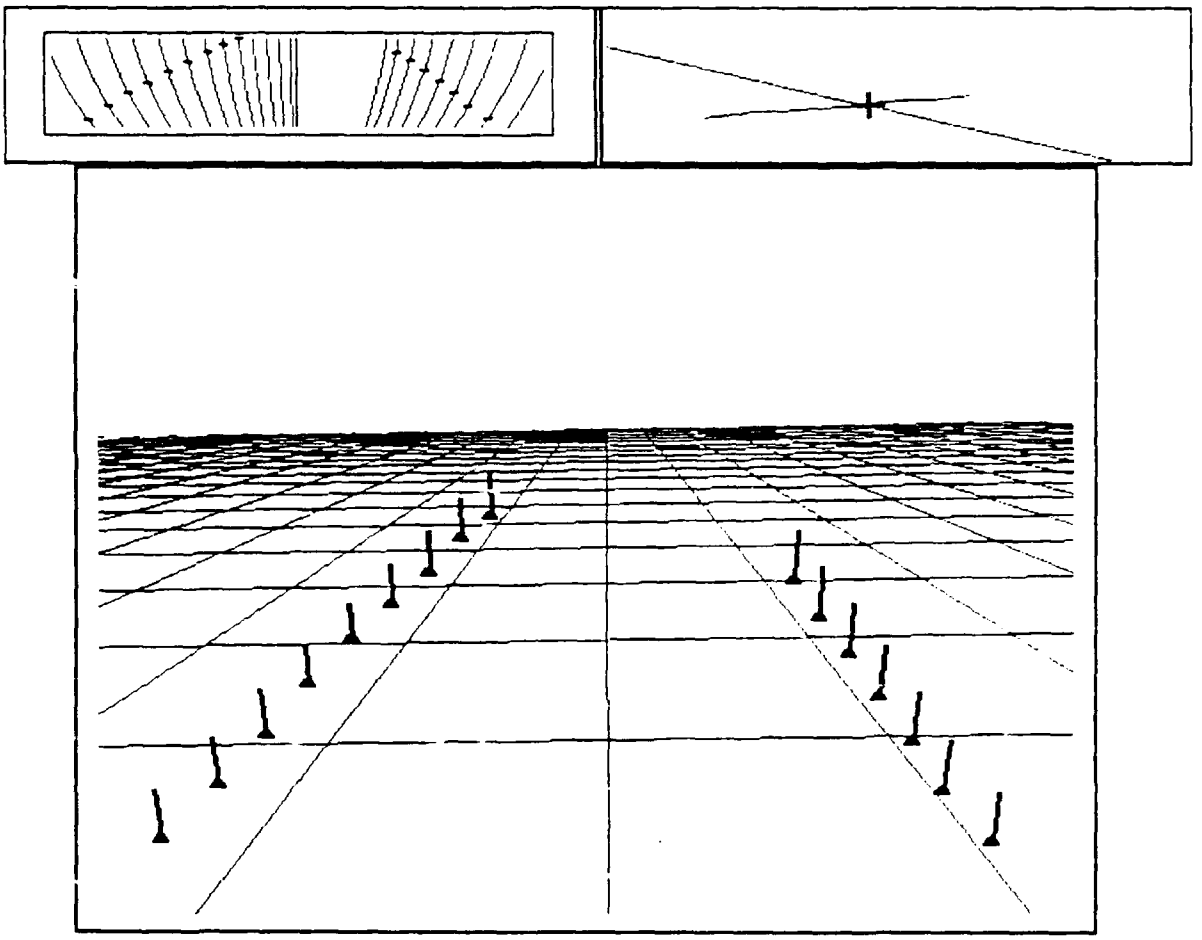


Figure 5. Results of range/video fusion algorithm showing (a) the epipolar arcs drawn within the ERIM image, (b) profiles showing the camera pixel ray and the elevation derived from the ERIM data along an epipolar arc, and (c) a perspective view of the resulting scene model.

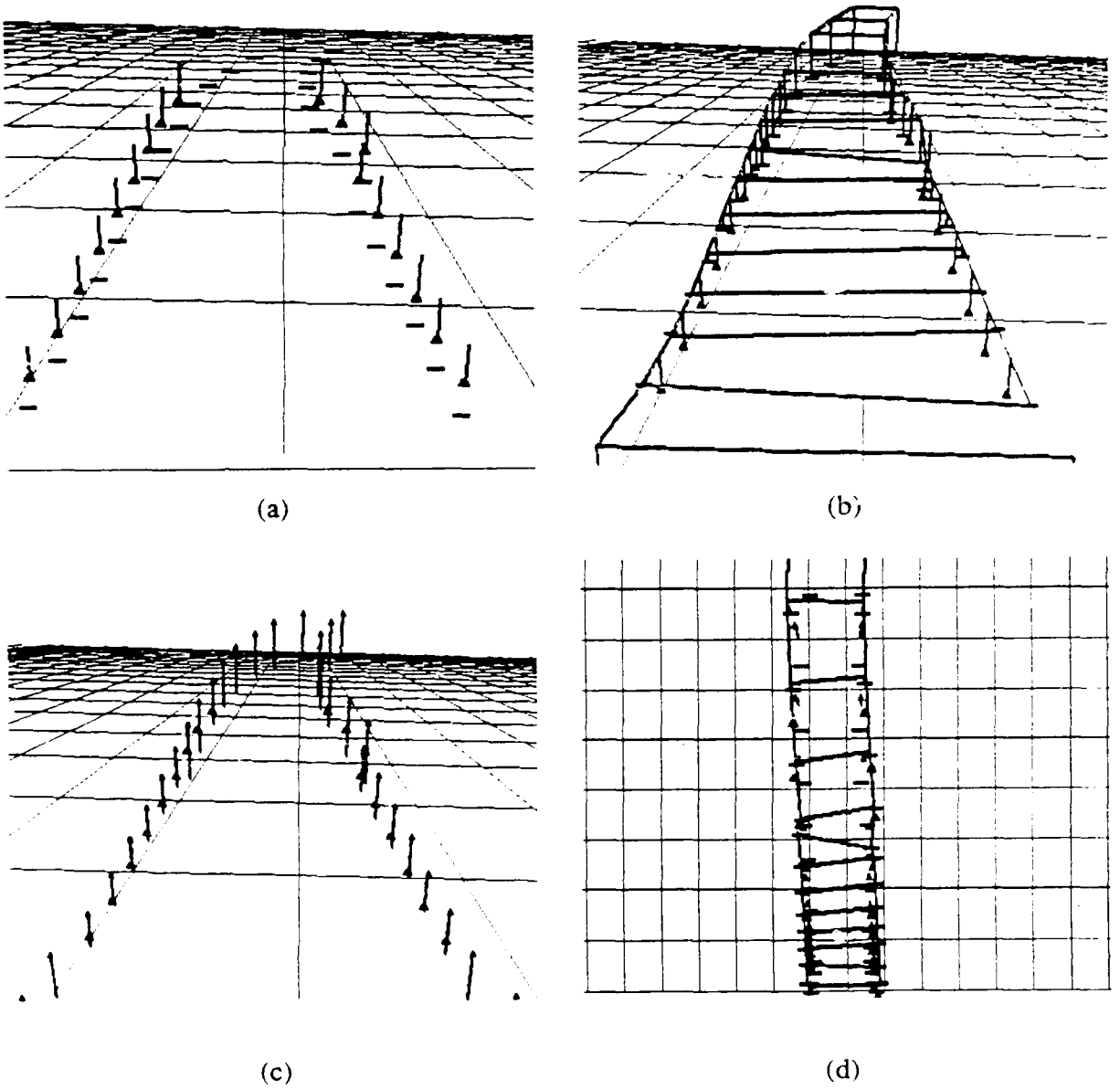


Figure 6. Results on a slightly curved, upsloping road scene. (a–c) show the fusion algorithm compared with: (a) the flat-earth algorithm, (b) the modified zero-bank algorithm, and (c) the hill-and-dale algorithm. An overhead view of all four approaches is shown in (d).

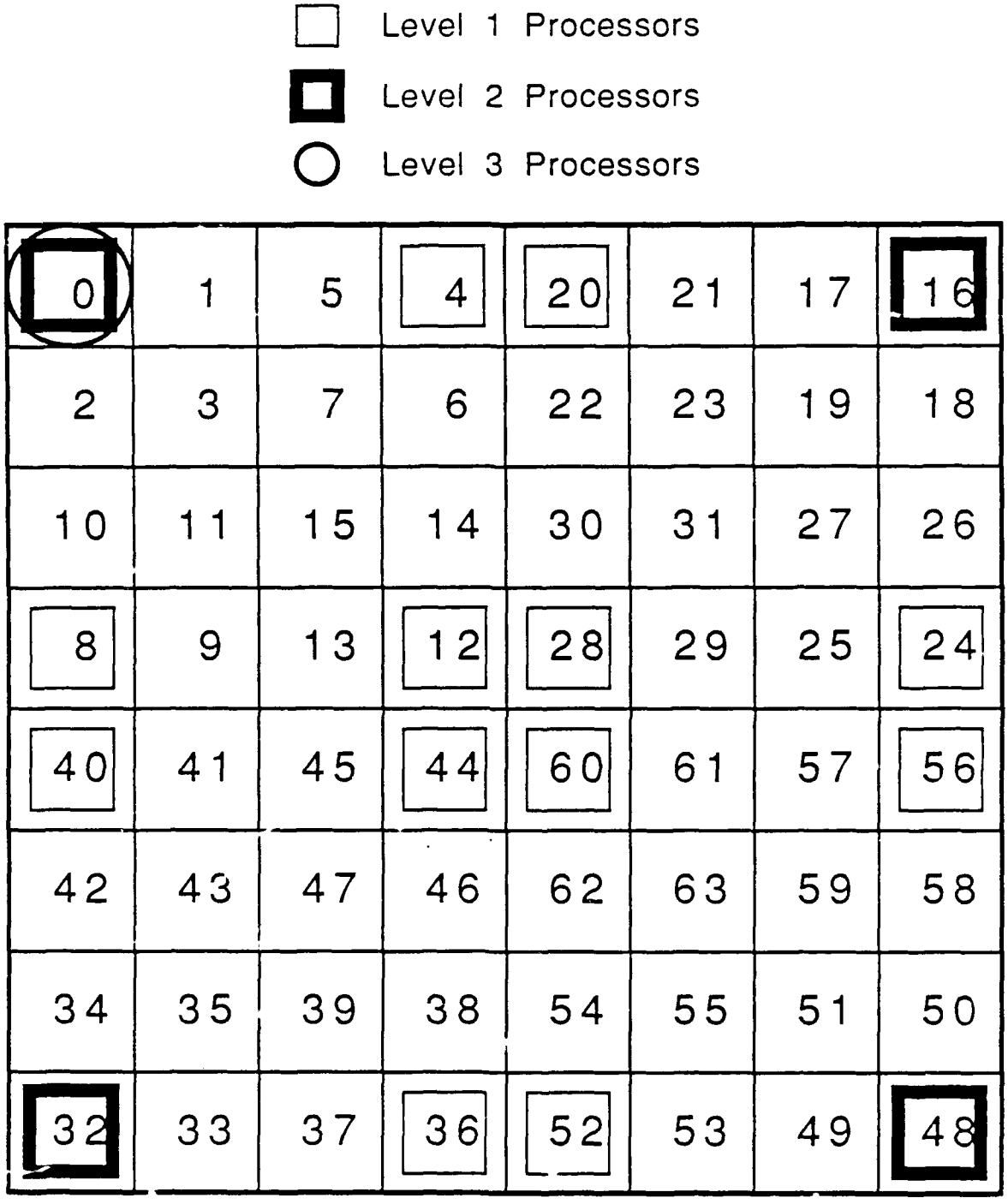


Figure 7. Layout of the nodes of a 4-level pyramid mapped onto virtual processors linked by a Boolean 6-cube.